(12)

FINAL TECHNICAL REPORT

for

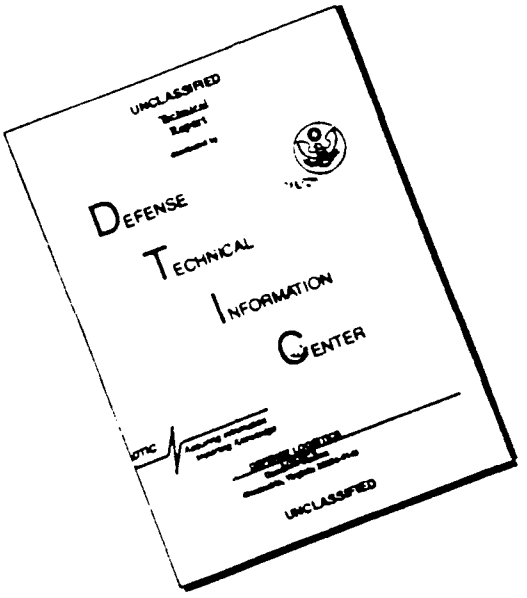
CONTRACT NUMBER N00014-88-k-0723

and project entitled

RECONFIGURATION SCHEMES FOR FAULT-TOLERANT PROCESSOR ARRAYS

Period:	88 Jul 01 to 92 Aug 31 (includes 1 year no-cost extension)
Scientific Officer:	Dr. Clifford Lau ONR Electronics Division
Prepared by:	Dr. José A. B. Fortes Purdue University
Date:	92 Oct 15

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

FINAL TECHNICAL REPORT

for

CONTRACT NUMBER N00014-88-k-0723

and project entitled

RECONFIGURATION SCHEMES FOR FAULT-TOLERANT PROCESSOR ARRAYS

Period:	88 Jul 01 to 92 Aug 31 (includes 1 year no-cost extension)
Scientific Officer:	Dr. Clifford Lau ONR Electronics Division
Prepared by:	Dr. José A. B. Fortes Purdue University
Date:	92 Oct 15

Dist A. per telecon Dr. C. Lau
ONR/Code 1114SE
Arlington, VA 22217-5000

11/20/92 CG

Distribution:

Addressee	*Number of Copies*
Dr. Clifford Lau Electronics Division Office of Naval Research (Code 1114SE) 800 N. Quincy Street Arlington, VA 22217-5000	1
Administrative Contracting Officer Office of Naval Research Resident Representative 536 South Clark Street, Rm 286 Chicago, IL 60605-1588	1
Director, Naval Research Laboratory ATTN: Code 2627 Washington, D.C. 20375	6
Defense Technical Information Center Building 5, Cameron Station Alexandria, VA 22314	12

Table of Contents

1. Introduction

This project addressed several aspects of the the problem of designing highly-reliable dynamically reconfigurable processor arrays. The proposed work focused mainly on reconfiguration schemes required to implement fault-tolerant processor arrays. According to the original statement of work, the following complementary objectives were pursued (relevant references to work performed under the grant appear in parentheses):

1. a methodology for the design and evaluation of *processor-switched* arrays ([2], [7], [8], [11], [12], [13], [24]),
2. a methodology for the design and evaluation of *multi-level hierarchically reconfigurable* processor arrays ([10],[23],[25]),
3. a methodology for the design of *fault-tolerant interconnection routers* for processor arrays with *decentralized routing* control ([5], [17], [19], [20], [22], [27]) and
4. *algorithm reconfiguration strategies* which, together with hardware reconfiguration schemes, can be used to achieve graceful degradation in processor arrays ([1], [3], [4], [6], [19], [15], [16], [26]).

The emphasis of the proposed research was on the development of *optimal* reconfiguration schemes for each of the above objectives by using mathematical and simulation tools. For this purpose, evaluation methods and adequate measures were also studied and developed. These measures include not only reliability but also joint measures of performance, hardware area and reliability. Software tools were developed to help in the process of evaluating the reconfiguration schemes proposed. The contributions of this work include a sound mathematical framework for the design of reliable reconfigurable processor arrays, tools for their evaluation and novel hardware designs for processor arrays and their components. These results will allow the design of very large processor array systems suited for applications characterized by the need to survive without maintenance in hard-to-predict harsh environments and long mission times. These areas of application include, among others, real-time computers, digital signal, image and speech processing systems, robotics, aerospace and transportation vehicles and remote data processing and sensing systems.

2. Annotated list of publications

The following publications report work funded in part by contract N00014-88-k-0723. After each reference a short explanation of its contents follows. For those references marked by an asterisk the full publication is included as in Section 3 of this report (Reprints of publications). The other references are to either earlier versions or conference versions of those marked by asterisks (most of which are archival journal papers).

[1]* Shang, W. and Fortes, J. A. B., "On the Optimality of Linear Schedules," *Journal of VLSI Signal Processing*, Volume 1, Number 3, November 1989, pp. 209-220.

Note - This paper shows how to compare the optimal linear schedule for a *single* output computed by a recurrence-like algorithm. These schedules and algorithms capture the majority of systolic array algorithms. It is also shown that optimal linear schedules are very close to the best possible schedules and, in many cases, just as good. These results are relevant to the problem of rescheduling systolic algorithms in arrays with faulty components.

[2]* Chean, M., and Fortes, J. A. B., "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays," *IEEE Computer*, December 1990, pp. 55-69.

Note- This paper surveys and classifies processor array reconfiguration schemes proposed by the authors and other researchers. Criteria for evaluation of the schemes and their relative merits are also discussed.

[3]* Shang, W., and Fortes, J. A. B., "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, Volume 40, Number 6, June 1991, pp. 723-743.

Note - This paper extends the results of reference [1] to the case when *several* outputs must be computed in the minimal amount of time.

[4]* Shang, W. and Fortes, J. A. B., "On the Independent Partitioning of Algorithms with Uniform Data Dependencies," *IEEE Transactions on Computers*, Volume 41, Number 2, February 1992, pp. 190-206.

Note - This paper presents a technique for optimal partition of algorithms into blocks of computations among which no communication is required. The same techniques can be used for the case when limited communication is possible. These techniques are potentially useful to reallocate computations in multiprocessor systems where failures result in diminished connectivity among processors.

[5]* Rau, D., Fortes, J. A. B. and Siegel, H. J., "Destination Tag Routing Techniques Based on a State Model for the IADM Network," *IEEE Transactions on Computers*, Volume 41, Number 3, March 1992, pp. 274-286.

Note - This paper proposes a simple destination-tag routing mechanism for a class of networks. It also shows how the message routing scheme supports the rerouting of messages around faulty nodes and links. These results are of use in fault-tolerant processor arrays where a multistage network is used to connect processors.

[6]* Shang, W. and Fortes, J. A. B., "On Mapping Uniform Dependence Algorithms into Lower Dimensional Processor Arrays," *IEEE Transactions on Parallel and Distributed Systems*, Volume 3, Number 3, May 1992, pp. 350-363.

Note - Two-dimensional processors with faulty elements can be easily reconfigured as one-dimensional arrays. However, it is necessary to remap the original algorithm into an array of smaller dimension than that of the original array. This paper presents a mapping methodology which addresses this problem.

[7]* Chean, M., and Fortes, J. A. B., "The Full-Use-of-Suitable-Spaces (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerance Processor Arrays," *IEEE Transactions on Computers,* Vol. 39, No. 4, April 1990, pp. 564-571.

Note - This paper describes a new technique for processor array reconfiguration and discusses its performance. It allows a processor array to reconfigure in the presence of up to as many faults as the number of spare processors available with a probability of more than 98%.

[8]* Lopez-Benitez, N. and Fortes, J. A. B., "Detailed Modeling and Reliability Analysis of Fault-Tolerant Processor Arrays," *IEEE Transactions on Computers.* Vol. 41, No. 9, September 1992, pp. 1193-1200.

Note - This paper presents an efficient approach to the problem of modeling and evaluating the reliability of fault-tolerant processor arrays. It also reports results obtained by using a software package that implements the proposed approach.

[9] O'Keefe, M. T., Fortes, J. A. B. and Wah, B. W., "On the Relationship Between Two Systolic Array Design Methodologies," to appear in *IEEE Transactions on Computers.*

Note - This paper extends earlier work by the same authors and shows how the understanding of the relations between two design methodologies was useful in deriving new systolic array designs. The techniques are of use to algorithm reconfiguration as well.

[10]* Wang, Y-X. and Fortes, J.A.B., "On the Analysis and Design of Hierarchical Fault-Tolerant Processor Arrays," International Workshop on Defect and Fault-Tolerance in VLSI Systems, October 6-7, 1988, Springfield, Massachusetts.

Note - This paper addresses the problem of designing hierarchically organized fault-tolerant processor arrays so that their reliability is optimized.

[11] Chean, M. and Fortes, J. A. B., "FUSS: A Reconfiguration Scheme for Fault-Tolerant Processor Arrays," International Workshop on Hardware Fault-Tolerance in Multiprocessors, June 19-20, 1989, Urbana, Illinois.

Note - this paper reports in "extended-abstract" form work that led to the results in [7].

[12] Lopez-Benitez, N. and Fortes, J. A. B., "Detailed Modeling of Fault-Tolerant Processor Arrays," 19th Int'l Symposium on Fault-Tolerant Computing, June 21-23, 1989, Chicago, Illinois, pp. 545-552.

Note - Extended conference version of [8].

[13]* Wang, Y.-X and Fortes, J.A.B., "Estimates of MTTF and Optimal Number of Spares of Fault-Tolerant Processor Arrays," 20th Int'l Symposium on Fault-Tolerant Computing, June 26-28, 1990, Newcastle Upon Tyne, U.K., pp. 184-191.

Note - This paper presents an efficient analytical approach to the evaluation of Mean-Time-To-Failure (MTTF) of fault-tolerant arrays. It also proposes a design procedure that optimizes MTTF.

[14] Shang, W., and Fortes, J.A.B., "Time-Optimal and Conflict-Free Mappings of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays," 1990 International Conference on Parallel Processing, St. Charles, Illinois, Vol. I, pp. 101-110.

Note - An earlier conference version of [6].

[15]* Shang, W., O'Keefe, M. T., and Fortes, J. A. B., "On Loop Transformations for Generalized Cycle Shrinking," 1991 International Conference on Parallel Processing, August, St. Charles, Illinois, Vol. II, pp. 132-141. Improved version to appear in IEEE Transactions on Parallel and Distributed Systems.

Note - This paper shows how certain compiler optimizations (generically denoted as cycle shrinking transformations) can be unified and generalized by linear schedules such as those used to schedule systolic algorithms.

[16] Shang, W., Yang, Z. and Fortes, J. A. B., "Conflict-Free Scheduling of Nested Loop Algorithms on Lower Dimensional Processor Arrays," Sixth International Parallel Processing Symposium, March 23-26, 1992, Beverly Hills, California.

Note - This paper improves upon the method proposed in [ ] by proposing a conflict-avoidance procedure with reduced computational complexity.

[17]* Cam, H. and Fortes, J. A. B., "Fault-Tolerant Self-Routing Permutation Networks," 1992 International Conference on Parallel Processing, August 17-21, St. Charles, Illinois, Vol. I, pp. 243-247.

Note - this paper proposes a novel fault-tolerant interconnection network capable of implementing any permutation using distributed destination-tag based self-routing of messages. The time needed to implement any permutation is the smallest of any network with comparable hardware complexity.

[18]* Saghi, G., Siegel, H. J., and Fortes, J. A. B., "On the Viability of a Quantitative Model of System Reconfiguration Due to a Fault," 1992 International Conference on Parallel Processing, August 17-21, St. Charles, Illinois, Vol. I, pp. 233-242.

Note - This paper proposes a quantitative model that can be used by an operating system to decide what reconfiguration strategy should be used. It discusses how the costs of three reconfiguration schemes may be quantifiable in a realistic setting.

[19]* Cam, H. and Fortes, J. A. B., "Frames: A Simple Characterization of Permutations realized by Frequently Used Networks," Tech. Rpt. TR-EE-92-32, School of Electrical Engineering, Purdue University, July 1992, 44 pages.

Note - This report proposes a simple characterization of permutation capabilities of interconnection networks. The proposed approach makes it computationally easy to detect whether a particular interconnection pattern (i.e., configuration) can be implemented by a network. Submitted for publication in IEEE Transactions on Computers.

[20]* Cam, H. and Fortes, J. A. B., "New Routing Algorithms for Benes and Reduced $\Omega_N\Omega_N^{-1}$ Networks," Submitted for publication in IEEE Transactions on Computers. *Note* - this paper presents algorithms for the routing of messages in two important rearrangeable networks. They can be used to set processor array systems into configurations that exclude faulty components.

[21] Lopez-Benitez, N. and Fortes, J. A. B., "Detailed Modeling and Reliability Analysis of Fault-tolerant Processor Arrays," Tech. Rpt. TR-EE-89-31, School of Electrical Engineering, Purdue University, June 1989, 94 pages.

Note - Long version of [8].

3. Thesis publications

[22] Rau, D., (December 1988) "Destination-Tag Routing Schemes for Multistage Interconnection Networks with Redundant Paths"

[23] Lopez-Benitez, N., (August 1989) Detailed Modeling and Reliability Estimation of Fault-Tolerant Processor Arrays

[24] Chean, M., (August 1989) Hardware Reconfiguration for Fault-Tolerant Processor Arrays

[25] Wang, Y-X., (May 1990) Approximate Evaluation of Reliability, Mean-Time-To-Failure and Optimal Redundancy of Fault-Tolerant Processor Arrays

[26] Shang, W., (May 1990) Scheduling, Partitioning and Mapping of Uniform Dependence Algorithms on Processor Arrays

[27] Cam, H. , (May 1992) Design and Permutation Routing Algorithms of Rearrangeable Networks

4. Graduate students supported

Darween Rau (male, Asian non-US)

Mengly Chean (male, Asian US)

Weijia Shang (female, Asian non-US)

Noe Lopez-Benitez (male, Hispanic US)

Mathew O'Keefe (male, US)

5. Invited Lectures of J. Fortes

[1] "Processor Arrays: Concept, Design, Applications and Real Machines," III Simposium Internacional Computacion UDEM'90, Universidad de Monterrey, Monterrey, Mexico, September 28, 1990.

[2] "Systolic Array Design in the Linear Algebra Framework," Minisymposium on Linear Algebra in Systolic Arrays, Second SIAM Conference on Linear Algebra, San Francisco, November 5-8, 1990.

[3] "Mapping Algorithms onto Parallel Architectures: Time Schedules," Second International IEE Specialist Seminar on "The Design and Application of Parallel Digital Processors," Lisbon, Portugal, April 15-19, 1991.

[4] "Generalized Cycle Shrinking," Conference on Algorithms and Parallel VLSI Architectures II, Chateau de Bonas, Gers, France, June 1991.

[5] "Exploiting Parallelism with Linear Schedules," International Conference for Young Computer Scientists, Beijing, China, July 1991.

[6] "Linear Schedules: Optimization and Relation to Parallelizing Compiler Techniques," Center for Supercomputing Research and Development, University of Illinois, February 4, 1992.

[7] "On the Adaptability of Algorithms and Architectures," CONPAR-VAPP 92 (1992 Conference on Parallel Processing - Vector and Array Parallel Processors), Lyon, France, September 1-4, 1992.

6. Honors and Editorial Positions of J. Fortes

1989 Guest Editor (with S. Y. Kung) of 2 special issues of *Journal of VLSI Signal Processing* on Systolic Systems.

Subject Area Editor for *Journal of Parallel and Distributed Computing*

1990-Present
Subject Area Editor for *Journal of Parallel and Distributed Computing*
Member of Editorial Board of *Journal of VLSI Signal Processing*

1992-Present
Member of Editorial Board of *IEEE Transactions on Parallel and Distributed Systems*

J. Fortes - IEEE Computer Society Distinguished Visitor (1991, 1992)

7. Reprints of Publications

REFERENCE NO. 1

Shang, W. and Fortes, J. A. B., "On the Optimality of Linear Schedules," *Journal of VLSI Signal Processing*, Volume 1, Number 3, November 1989, pp. 209-220.

Note - This paper shows how to compare the optimal linear schedule for a *single* output computed by a recurrence-like algorithm. These schedules and algorithms capture the majority of systolic array algorithms. It is also shown that optimal linear schedules are very close to the best possible schedules and, in many cases, just as good. These results are relevant to the problem of rescheduling systolic algorithms in arrays with faulty components.

Journal of VLSI Signal Processing, 1, 209–220 (1989)

On the Optimality of Linear Schedules

WEIJIA SHANG AND JOSE A.B. FORTES
School of Electrical Engineering, Purdue University, West Lafayette, IN 47907

Received November 8, 1988

Abstract. An algorithm can be modeled as a set of indexed computations, and a schedule is a mapping of the algorithm index space into time. *Linear schedules* are a special class of schedules that are described by a linear mapping and are commonly used in many systolic algorithms. *Free schedules* cause computations of an algorithm to execute as soon as their operands are available. If one computation uses data generated by another computation, then a data dependence exists between these two computations which can be represented by the difference of their indices (called *dependence vector*). Many important algorithms are characterized by the fact that data dependencies are *uniform*, i.e., the values of the dependence vectors are independent of the indices of computations. There are applications where it is of interest to find an optimal linear schedule with respect to the time of execution of *a specific* computation of the given algorithm. This paper addresses the problem of identifying optimal linear schedules for uniform dependence algorithms so that the execution time of *a specific* computation of the algorithm is minimized and proposes a procedure to solve this problem based on the mathematical solution of a linear optimization problem. Also, linear schedules are compared with free schedules. The comparison indicates that optimal linear schedules can be as efficient as free schedules, the best schedules possible, and identifies a class of algorithms for which this is always true.

1. Introduction

The algorithms under consideration in this paper are characterized by *uniform data dependencies* and *unit-time computations*; they include those described by single uniform recurrences [1] and resemble a very large number of systolic computations and algorithms described by programs with nested loops. Informally, an algorithm is represented in this paper as a partially ordered subset of a multidimensional integer lattice (called *index set*). The points of this lattice correspond to (i.e., are the indices of) computations, and the partial order reflects the data dependencies between them. These data dependencies are represented as vectors that connect points of the lattice. If a given dependence vector is always present when the vector difference between any two lattice points equals the dependence vector, then the dependence is said to be *uniform*. If all dependences are uniform then the algorithm is said to be a *uniform dependence algorithm*.

This research was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contracts No. 00014-85-k-0588 and No. 00014-88-k-0723.

A *linear schedule* is a mapping from the multidimensional algorithm index set into the one-dimensional time space; this mapping is expressed as a linear transformation that involves the multiplication of a vector, called *linear schedule vector*, by each and every point of the index set. The image of the index point under the mapping is the time of execution of the computation indexed by that point. This algorithm model and the notion of linear schedule are easily related to similar models and concepts used in [1]–[13] and several other works.

There are two formulations of optimal linear schedule problem. In one case, the execution time of the set of all computations of the algorithm is to be minimized. In [14], a procedure is provided to find an optimal solution for this problem for any uniform dependence algorithms whose index sets are convex polyhedra. In some applications, given an algorithm, it is of interest to find an optimal linear schedule with respect to the execution time of one specific computation. Clearly, to execute this specific computation, all computations on which that specific computation depends must execute first. This gives the second formulation of optimal linear schedule problem where the goal is to minimize the execution time of a specific computation of the algorithm.

A *free schedule* schedules computations to execute as soon as their operands are available. The total execution time that results from using a free schedule is an exact lower bound for the execution time of the algorithm. In [1] and [12] the execution times achievable by linear schedules and free schedules are compared. The difference between the execution time achieved by the free schedule and the execution time achievable by an optimal linear schedule is bounded by a constant [1]. In [12], it was found that the difference is equal to either one or zero for a set of 25 algorithms.

In this paper, the second formulation of the linear schedule problem is considered and a procedure is proposed to find the linear schedule that minimizes the execution time of *a specific* computation of the algorithm. Also, the results reported in [1] and [12] on the comparison of linear schedules and free schedules are extended. A class of algorithms is identified for which the difference between the execution times achieved by free schedules and linear schedules, respectively, is always zero. This means that linear schedules can achieve the execution time lower bound for this class of algorithms.

This paper is organized as follows. The basic terminology and definitions used throughout the paper are introduced in Section 2 as well as the formulation of the problem of determining an optimal linear schedule when the execution time of a single specific computation is to be minimized. Section 3 is dedicated to the solution of this problem. Linear schedules are compared to free schedules in Section 4, and sufficient conditions are provided for an algorithm to have an optimal linear schedule which has the same execution time as the free schedule. Section 5 is dedicated to conclusions.

2. Terminology and Definitions

Throughout this paper, sets, matrices and row vectors are denoted by capital letters, column vectors are represented by lower case symbols with an overbar, and scalars correspond to lower case letters. The transposes of a vector $\bar{v}$ and a matrix M are denoted $\bar{v}^T$ and M^T, respectively. The symbol E_i denotes the row vector whose entries are all zeroes except that the *ith* entry is equal to unity. The vector $\bar{1}$ (or $\bar{0}$) denotes the row vector or column vector whose entries are all ones (or zeroes). The dimensions of vectors $\bar{1}$ and $\bar{0}$ and whether they denote row or column vectors are implied by the context in which they are used. The symbol I denotes the identity matrix. The rank of a matrix A is denoted *rank*(A). The set of rational numbers, the real space and the set of integers are denoted Q, IR and Z, respectively. The set of non-negative integers and the set of positive integers are denoted N and N^+, respectively. The empty set is denoted $\mathcal{D}$. The symbol $\lfloor a \rfloor$ denotes the greatest integer that is less than or equal to a. The greatest common divisor of integers $a_1, \ldots, a_n$ is denoted $\gcd(a_1, \ldots, a_n)$. As a final remark, if x is an element of a set S, the notation $x \in S$ is used and this notation is "abused" to indicate also that a column vector $\bar{m}_j$ (or row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j \in M$ ($M_i \in M$) means $\bar{m}_j$ (M_i) is a column (row) vector of matrix M.

The algorithms of interest in this paper are the so-called uniform dependence algorithms defined as follows.

Definition 2.1. (Uniform dependence algorithm): A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\bar{j}) = g_{\bar{j}}(v(\bar{j} - \bar{d}_1), v(\bar{j} - \bar{d}_2), \ldots, v(\bar{j} - \bar{d}_m)) \tag{2.1}$$

where

1. $\bar{j} \in J \subset Z^n$ is an index point (a column vector), J is the *index set* of the algorithm, and $n \in N^+$ is the number of components of $\bar{j}$;
2. $g_{\bar{j}}$ is the computation indexed by $\bar{j}$, i.e., a single-valued function computed "at point $\bar{j}$" in a *single unit of time*;
3. $v(\bar{j})$ is the value computed "at $\bar{j}$," i.e., the result of computing the right hand side of (2.1) and
4. $\bar{d}_i \in Z^n$, $i = 1, \ldots, m$, $m \in N$ are *dependence vectors*, also called *dependencies*, which are constant (i.e., independent of $\bar{j} \in J$); the matrix $D = [\bar{d}_1, \ldots, \bar{d}_m]$ is called the *dependence matrix*.

The class of uniform dependence algorithms is a simple extension of the class of computations described by uniform recurrence equations [1]. The main difference is that uniform dependence algorithms allow for different functions to be computed (in a unit of time) at different points of the index set. From a practical viewpoint, uniform dependence algorithms can be easily related to programs where (1) a single statement appears in the body of a multiply nested loop and (2) the indices of the variable in the left-hand side of the statement differ by a constant from the corresponding indices in each reference to the same variable in the right-hand side. Alternative computations can occur in each iteration as a result of a single conditional statement as long as data dependencies do not change. Nested loop programs

with multiple statements can also use the techniques of this paper together with the alignment method discussed in [15] and [16].

For the purpose of this paper, only structural information of the algorithm, i.e., the index set J and the dependence matrix D, is needed. Other information, such as what computations occur at different points and where and when input/output of variables takes place, can be ignored. Therefore, a uniform dependence algorithm with index set J and dependence matrix D is here on characterized simply by the pair (J, D). It is also assumed that, as in Definition 2.1, the letters n and m always denote the dimension of index points in J and the number of dependence vectors, respectively. The following example illustrates the concept of uniform dependence algorithms.

Example 2.1. Consider the following uniform dependence algorithm:

$$x(j_1, j_2) = g(x(j_1 - 1, j_2 + 3), x(j_1 - 2, j_2 - 4), x(j_1 - 2, j_2))$$

where

$$j_1 = 0, \ldots, 2s$$

$$j_2 = \begin{cases} 0, \ldots, 2s - j_1 & \text{if } s \le j_1 \le 2s \\ s - j_1, \ldots, 2s - j_1 & \text{if } 0 \le j_1 \le s \end{cases}, \quad s \in N^+$$

The index set J of this algorithm is shown in Figure 1. The reader can verify easily that it can be described as $J = \{\bar{j} = [j_1, j_2]^T : A\bar{j} \le \bar{b}, \bar{j} \in Z^2\}$ where

$$A = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} 0 \\ 0 \\ 2s \\ -s \end{bmatrix}$$

The dependence matrix is $D = [\bar{d}_1, \bar{d}_2, \bar{d}_3]$ where $\bar{d}_1 = [1, -3]^T$, $\bar{d}_2 = [2, 4]^T$ and $\bar{d}_3 = [2, 0]^T$.

The dependence vectors $\bar{d}_i$, $i = 1, \ldots, m$ induce a partial ordering on the index set J. This ordering, denoted $\propto$, can be easily described in terms of the algorithm precedence graph. Informally, the set of nodes of the precedence graph is the index set, and if computation indexed by $\bar{j}_1$ depends directly on computation indexed by $\bar{j}_2$, then there is a directed edge from node $\bar{j}_2$ to node $\bar{j}_1$. The formal definition is given next.

Definition 2.2. (Algorithm precedence graph and directed path): The *precedence graph* of algorithm (J, D) is the directed graph (J, E) where J is the set of nodes of the graph and $E = \{(\bar{j}, \bar{j}') : \bar{j}' - \bar{j} = \bar{d}_i, \bar{d}_i \in D, \bar{j}, \bar{j}' \in J\}$ is the set of edges. If $(\bar{j}, \bar{j}') \in E$, then $\bar{j}$ and $\bar{j}'$ are called the tail and the head of edge $(\bar{j}, \bar{j}')$, respectively. If there exists a set of index points $\bar{j}_1, \ldots, \bar{j}_l$ such that $(\bar{j}, \bar{j}_1), (\bar{j}_1, \bar{j}_2), \ldots, (\bar{j}_l, \bar{j}^*) \in E$, then $\bar{j}$ is *connected* to $\bar{j}^*$. The tuple $L = (\bar{j}, \bar{j}_1, \ldots, \bar{j}_l, \bar{j}^*)$ is called a *directed path* and its *length* is the number of edges on the path, i.e., $l + 1$.

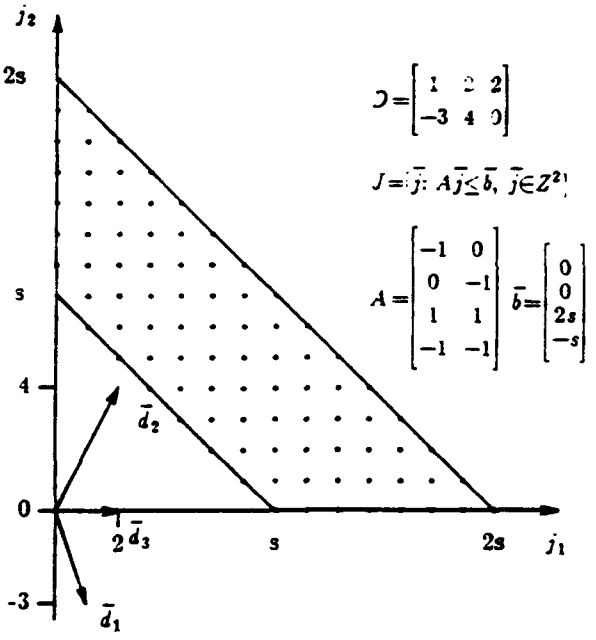


Fig. 1. The index set of the algorithm of Example 2.1.

The partial ordering $\propto$ induced by the dependence vectors on the index set J is such that, if $\bar{j}, \bar{j}' \in J$, then $\bar{j} \propto \bar{j}'$ if and only if there exists a directed path from $\bar{j}$ to $\bar{j}'$ in the precedence graph of the algorithm.

Definition 2.3. (Schedules): A *schedule* for algorithm (J, D) is a function $\sigma: J \to Z$ which is strictly monotone increasing with respect to the ordering $\propto$ induced by $\bar{d}_i$, $i = 1, \ldots, m$ in J, i.e., if $\bar{j} \propto \bar{j}'$ (computation indexed by $\bar{j}'$ depends on computation indexed by $\bar{j}$), then $\sigma(\bar{j}) < \sigma(\bar{j}')$.

In other words, a schedule is a mapping which assigns a time of execution to each computation of the algorithm in such a way that dependencies are preserved, i.e., if the computation indexed by $\bar{j}'$ depends on the computation indexed by $\bar{j}$, then computation indexed by $\bar{j}'$ can be executed only after the execution of computation indexed by $\bar{j}$. Examples of schedules of interest in this paper include free schedules and linear schedules, both of which are defined next.

Definition 2.4. *(Free schedule):* For algorithm (J, D) the *free schedule* is a mapping σ_f: $J \rightarrow Z$ such that

$$\sigma_f(\bar{j}) = \begin{cases} 0 & \text{if } \forall \bar{d}_i \in D,\ \bar{j}-\bar{d}_i \notin J \\ \max\{\sigma_f(\bar{j}-\bar{d}_i): \bar{j}-\bar{d}_i \in J,\ i=1, \ldots, m\}+1 & \text{if } \exists \bar{d}_i \in D,\ \bar{j}-\bar{d}_i \in J \end{cases}$$

Definition 2.5. For algorithm (J, D) a schedule σ_Π: $J \rightarrow Z$ *is a linear schedule* if $\sigma_\Pi(\bar{j}) = \lfloor \Pi\bar{j} + c \rfloor$, $\bar{j} \in J$, where $\Pi \in Q^{1\times n}$ is such that $\min\{\Pi\bar{d}_i: \bar{d}_i \in D\} = 1$ and $c = -\min\{\Pi\bar{j}: \bar{j} \in J\}$.

Definitions 2.3 and 2.4 are similar to equivalent definitions in [1]. Free schedules are the fastest schedules possible, and the total execution time achieved by free schedules is the exact lower bound of the execution time of any algorithm. It is relatively simple to verify that both free schedules and linear schedules satisfy Definition 2.3, i.e., they preserve data dependencies. In the last definition, the entries of Π can be any rational number. However, given such a Π, it is always possible to find $\Pi' \in Z^{1\times n}$ and a constant $disp\Pi' \in N^+$ such that $\Pi = \Pi'/disp\Pi'$ where $disp\Pi'$ equals the least common multiple of the denominators of the entries of Π. For this reason, the following definition of a linear schedule is equivalent to that of Definition 2.5 and is used throughout this paper.

Definition 2.6. *(Linear schedule and linear schedule vector):* For algorithm (J, D) a *linear schedule* is a function σ_Π: $J \rightarrow N$ such that

$$\sigma_\Pi(\bar{j}) = \lfloor (\Pi\bar{j} + c)/disp\Pi \rfloor, \quad \bar{j} \in J \tag{2.2}$$

where $\Pi \in Z^{1\times n}$, $disp\Pi = min\{\Pi\bar{d}_i: \bar{d}_i \in D\} > 0$, $gcd(\pi_1, \ldots, \pi_n) = 1$ and $c = -\min\{\Pi\bar{j}: \bar{j} \in J\}$. The row vector Π is called *linear schedule vector* associated with σ_Π.

In some applications, given an algorithm (J, D), it is of interest to find an optimal linear schedule with respect to the time of execution of one specific computation. Let $\bar{\rho}$ be a point in J. If there exists a directed path from point $\bar{j}$ to $\bar{\rho}$ in the precedence graph of the algorithm (J, D), $\bar{j}$ is a *predecessor* of $\bar{\rho}$. Clearly, to execute the computation indexed by $\bar{\rho}$, all the computations indexed by the predecessors of $\bar{\rho}$ have to be executed first. Let $\bar{j}_1, \ldots, \bar{j}_l$ be all the predecessors of $\bar{\rho}$. $H(\bar{\rho})$ be the set of $\bar{j}_1, \ldots, \bar{j}_l$ and $\bar{\rho}$ and D_ρ be the matrix of dependence vectors that are present in $H(\bar{\rho})$, i.e., $\bar{d}_i \in D_\rho$ if and only if there exist two points $\bar{j}_1$, $\bar{j}_2 \in H(\bar{\rho})$ such that $\bar{j}_1 = \bar{j}_2 + \bar{d}_i$. Suppose that there are m_ρ dependence vectors in D_ρ. Clearly, D_ρ is a submatrix of D and $H(\bar{\rho})$ is a subset of J, i.e., $(H(\bar{\rho}), D_\rho)$ is a sub-algorithm of (J, D). Due to the fact that the computation indexed by $\bar{\rho}$ is the last to execute, the total execution time t_ρ for $(H(\bar{\rho}), D_\rho)$ with a linear schedule τ_Π can be expressed as

$$t_\rho = \left\lfloor \frac{\max\{\Pi(\bar{\rho} - \bar{j}): \bar{j} \in H(\bar{\rho})\}}{\min\{\Pi\bar{d}_i: \bar{d}_i \in D_\rho\}} \right\rfloor + 1 \tag{2.3}$$

Because t_ρ is minimum if the argument of the floor function in Equation 2.3 is minimized, the problem of finding an optimal linear schedule with respect to the execution of one specific computation is formulated as follows:

Problem 2.1. (Time optimal linear schedule problem for execution of one computation): For algorithm (J, D), let $\bar{\rho} \in J$. The *time optimal linear schedule problem for execution of the computation indexed by* $\bar{\rho}$ consists of finding a linear schedule vector $\Pi_\rho \in Q^{1\times n}$ such that it minimizes

$$f_\rho = \frac{\max\{\Pi(\bar{\rho} - \bar{j}): \bar{j} \in H(\bar{\rho})\}}{\min\{\Pi\bar{d}_i: \bar{d}_i \in D_\rho\}} \tag{2.4}$$

$$\textit{subject to} \quad \Pi\bar{d}_i > 0 \quad \bar{d}_i \in D_\rho.$$

Notice that if Π_ρ is an optimal solution of Problem 2.1, so is $\alpha\Pi_\rho$ for any non-zero constant α. This guarantees that the optimal solution Π_ρ can always be obtained such that $\Pi_\rho \in Z^{1\times n}$ and the greatest common divisor of the components of Π_ρ is equal to one. After Π_ρ is found, then, according to Definition 2.6, the constant c can be determined and the corresponding optimal linear schedule σ_{Π_ρ} can be specified completely.

The solution of Problem 2.1 is discussed in the next section. From here on, the letters m_ρ and m'_ρ always denote the number of dependence vectors in matrix D_ρ and the rank of matrix D_ρ, respectively; and without loss of generality, it is assumed that D_ρ contains the first m_ρ columns of D, i.e., $D_\rho = [\bar{d}_1, \ldots, \bar{d}_{m_\rho}]$.

3. Solution of Time Optimal Linear Schedule Problem for Execution of One Computation

This section discusses the solution of Problem 2.1, i.e., the problem of finding an optimal linear schedule such that the execution time of the computation indexed by a point $\bar{\rho}\pi \in J$ is minimized. A theorem and a procedure are provided that describe how to find the solution of Problem 2.1. The long proof of the theorem is provided in Appendix. However, the idea of the proof is

described informally and examples are given to illustrate the main concepts.

Let $D(c_1 \ldots c_x/r_1 \ldots r_y)$ denote the submatrix of D containing the elements in columns $c_1, \ldots, c_x$ and rows $r_1, \ldots, r_y$, i.e., it contains the elements of D at the intersections of columns $c_1, \ldots, c_x$ and rows $r_1, \ldots, r_y$, respectively. If $D(c_1 \ldots c_k/r_1 \ldots r_k)$ is nonsingular, an integer row vector $V = [v_1, \ldots, v_k] \in Z^{1\times k}$ is defined as $V = \beta \vec{1} D^{-1}(c_1 \ldots c_k/r_1 \ldots r_k)$ where β is a positive integer constant such that $gcd(v_1, \ldots, v_k) = 1$. In other words, V is a vector whose entries are the sums of the corresponding columns of $D^{-1}(c_1 \ldots c_k/r_1 \ldots r_k)$ scaled so that they are integers with the greatest common divisor equal to the unity. If $D(c_1 \ldots c_k/r_1 \ldots r_k)$ is nonsingular, then $\Pi(c_1 \ldots c_k/r_1 \ldots r_k) = VB \in Z^{1\times n}$, where

$$B = \begin{bmatrix} E_{r_1} \\ \ldots \\ E_{r_k} \end{bmatrix}.$$

In words, the subvector $[\pi_{r_1}, \ldots, \pi_{r_k}]$ of $\Pi(c_1 \ldots c_k/r_1 \ldots r_k)$ is the same as V and the remaining entries of $\Pi(c_1 \ldots c_k/r_1 \ldots r_k)$ are zero. Finally, C^ρ denotes the set $\{\Pi(c_1 \ldots c_{m'_\rho}/r_1 \ldots r_{m'_\rho}): 1 \le c_1 < \ldots < c_{m'_\rho} \le m_\rho, 1 \le r_1 < \ldots < r_{m'_\rho} \le n\}$. The following example illustrates the notation and concepts just introduced.

Example 3.1. Consider the algorithm (J, D) of Example 2.1 where index set J is shown in Figure 1 and

$$D = \begin{bmatrix} 1 & 2 & 2 \\ -3 & 4 & 0 \end{bmatrix}.$$

According to the definitions just introduced, $D(1/1) = [1]$ (contains the element at the intersection of column 1 and row 1 of D), its corresponding $V = \beta \vec{1} D^{-1}(1/1) = [1]$ where $\beta = 1$, $B = E_1 = [1, 0]$ and $\Pi(1/1) = VB = [1, 0]$; similarly, $D(1/2) = [-3]$, its corresponding $V = \beta \vec{1} D^{-1}(1/2) = [-1]$ where $\beta = 3$, $B = E_2 = [0, 1]$ and $\Pi(1/2) = VB = [0, -1]$; and

$$D(12/12) = \begin{bmatrix} 1 & 2 \\ -3 & 4 \end{bmatrix},$$

its corresponding $V = \beta \vec{1} D^{-1} = [7, -1]$ where $\beta = 10$, $B = I$ and $\Pi(12/12) = VB = V = [7, -1]$.

Without loss of generality let $D_\rho = [\vec{d}_1, \ldots, \vec{d}_{m_\rho}]$, $rank(D_\rho) = m'_\rho \le m_\rho$. The next theorem states that C^ρ is the candidate set for $(H(\vec{\rho}), D_\rho)$, i.e., $\Pi_\rho \in C^\rho$.

THEOREM 3.1. *Consider algorithm (J, D) and let $\vec{\rho} \in J$. An optimal solution Π_ρ of Problem 2.1 for $(H(\vec{\rho}), D_\rho)$ belongs to C^ρ, i.e.,*

$$\Pi_\rho \in C^\rho = \{\Pi(c_1 \ldots c_{m'_\rho}/r_1 \ldots r_{m'_\rho}): 1 \le c_1 < \ldots < c_{m'_\rho} \le m_\rho, \quad 1 \le r_1 < \ldots < r_{m'_\rho} \le n\} \tag{3.1}$$

Proof. Provided in Appendix.

The idea of the proof is as follows. The fact that the computation indexed by $\vec{\rho}$ is the last to execute is explored and Problem 2.1 is reformulated as a linear programming problem. Its dual is considered. For $m'_\rho = n$, it is shown that $G = [\vec{d}_{c_1}, \ldots, \vec{d}_{c_n}]$, where $\vec{d}_{c_1}, \ldots, \vec{d}_{c_n} \in D_\rho$, is an optimal basis for the dual problem and $\vec{1} G^{-1} = \Pi_\rho$ is an optimal solution of Problem 2.1. So $\Pi_\rho = \Pi(c_1 \ldots c_n/1 \ldots n)$ and it belongs to C^ρ. If $m'_\rho < n$, $H(\vec{\rho})$ is transformed into an m'_ρ-dimensional space by a bijective linear mapping τ specified by a matrix $T \in Z^{m'_\rho \times n}$. By applying the same reasoning in the case where $m'_\rho = n$, it is shown that $\Pi_\rho = \Pi(c_1 \ldots c_{m'_\rho}/r_1 \ldots r_{m'_\rho})$ for some values of $r_1, \ldots, r_{m'_\rho}$. The candidate set C^ρ is constructed by the following procedure.

Procedure 3.1. (Construction of C^ρ):

Input: Algorithm $(H(\vec{\rho}), D_\rho)$

Output: A finite candidate set C^ρ containing the optimal solution of Problem 2.1.

Step 1: $C^\rho = \emptyset$ and $l = 1$.

Step 2: Pick up a set of m'_ρ linearly independent columns from D_ρ.

Step 3: Let the set of m'_ρ linearly independent columns being processed be $\vec{d}_{c_1}, \ldots, \vec{d}_{c_{m'_\rho}}$. Find a matrix

$$B = \begin{bmatrix} E_{r_1} \\ \ldots \\ E_{r_{m'_\rho}} \end{bmatrix}, \quad 1 \le r_1 < \ldots < r_{m'_\rho} \le n,$$

such that $D(c_1 \ldots c_{m'_\rho}/r_1 \ldots r_{m'_\rho}) = B[\vec{d}_{c_1}, \ldots, \vec{d}_{c_{m'_\rho}}]$ is not singular and set $\Pi_l = \Pi(c_1 \ldots c_{m'_\rho}/r_1 \ldots r_{m'_\rho})$. $C^\rho = C^\rho \cup \{\Pi_l\}$, $l = l + 1$.

Step 4: Check if all distinct sets of m'_ρ linearly independent columns from D_ρ have been processed. If not, pick up an unprocessed set and go to Step 3. Otherwise, stop.

Clearly, Step 3 dominates the whole procedure. For Step 3, each iteration needs at most $O(\binom{n}{m'_\rho} m'^3_\rho)$ operations to find matrix B such that $B[\vec{d}_{c_1}, \ldots, \vec{d}_{c_{m'_\rho}}]$ is non-singular and there are at most $\binom{m_\rho}{m'_\rho}$ iterations. So

the complexity of Procedure 3.1 is bounded above by $O(\binom{n}{m'_\rho}\binom{m_\rho}{m'_\rho}m'^3_\rho)$ and is independent of the size of the algorithm (i.e., does not depend on the cardinality of $H(\bar{\rho})$). If $m'_\rho = m' = n$, then the complexity is $O(n^3)$. The following example shows how to construct C^ρ according to Procedure 3.1.

Example 2.2. Consider the algorithm of Example 2.1 where

$$D = \begin{bmatrix} 1 & 2 & 2 \\ -3 & 4 & 0 \end{bmatrix}.$$

Figure 2 shows the index set J when $s = 6$. Let $\bar{\rho} = [11, 0]^T$. Then $H(\bar{\rho}) = \{[11, 0]^T, [9, 0]^T, [7, 0]^T, [8, 3]^T, [6, 3]^T, [4, 3]^T, [5, 6]^T, [3, 6]^T, [1, 6]^T, [2, 9]^T, [0, 9]^T\}$. The set $H(\bar{\rho})$ contains points marked by "■" in figure 2. Only dependence vectors $[1, -3]^T$ and $[2, 0]^T$ are present in $H(\bar{\rho})$, i.e.,

$$D_\rho = \begin{bmatrix} 1 & 2 \\ -3 & 0 \end{bmatrix}.$$

There is only one combination of two dependence vectors from D_ρ which determines a vector $\Pi(13/12) = [3, -1]$. Therefore, $C^\rho = \{[3, -1]\}$ and $\Pi_\rho = [3, -1]$. According to [14], the optimal linear schedule for (J, D) is $\Pi(12/12)$ which is different from Π_ρ. This implies that, in general, the optimal linear schedule for (J, D) is not necessarily the same as the one for $(H(\bar{\rho}), D_\rho)$ where $\bar{\rho} \in J$. The total execution time for $(H(\bar{\rho}), D_\rho)$ by Π_ρ is $t(\Pi_\rho) = \sigma_{\Pi_\rho}(\bar{\rho}) - \sigma_{\Pi_\rho}([0, 9]^T) + 1 = 8$. Notice that the total execution time by the free schedule for $(H(\bar{\rho}), D_\rho)$ is also 8 units, i.e., the total execution time by the optimal linear schedule σ_{Π_ρ} is equal to the total execution time by the free schedule. So, the optimal linear schedule achieves the lower bound of the execution time of $(H(\bar{\rho}), D_\rho)$ in this example.

4. Comparison of Execution Times by Optimal Linear Schedules and Free Schedules

This section compares the execution time achievable by optimal linear schedules with that of free schedules. First, a sufficient condition on the dependence matrix D of an algorithm is provided which guarantees that both schedules achieve the same execution times for a specific computation of the algorithm. Afterwards, the execution time of all computations of the algorithm is briefly considered. It is shown by examples that, for the last case, both the size and shape of the index set

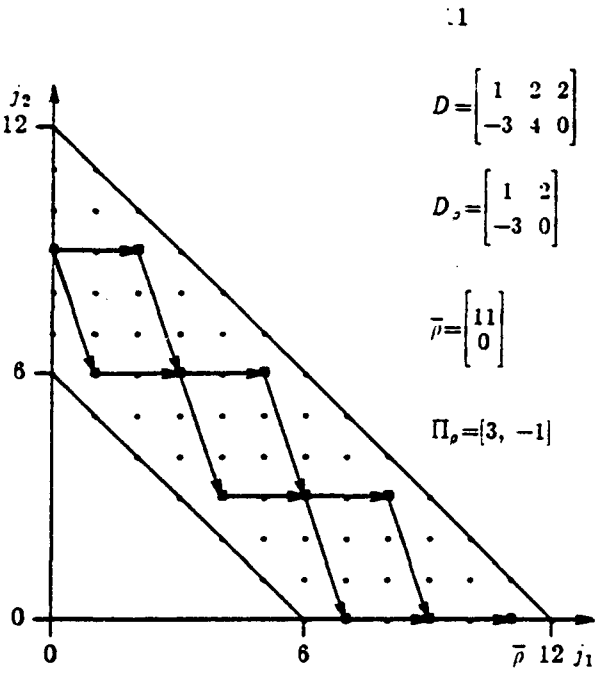


Fig. 2. The index set of the algorithm of Example 3.1 when $s = 6$. $H(\bar{\rho})$ is the set of points which are marked by "■" and connected to $\bar{\rho}$ by dependence vectors $[1, -3]^T$ and $[2, 0]^T$.

determine how linear schedules compare with free schedules.

Let $t(\Pi, J, D)$, $t(\sigma_f, J, D)$ denote the total execution time for algorithm (J, D) achieved by the linear schedule σ_Π and by the free schedule σ_f, respectively. Clearly, if l equals the length of the longest path in (J, E), the algorithm precedence graph of algorithm (J, D) defined in Definition 2.2, then $t(\sigma_f, J, D) = l + 1$. Let $k(H(\bar{\rho}), D_\rho) = t(\Pi_\rho, H(\bar{\rho}), D_\rho) - t(\sigma_f, H(\bar{\rho}), D_\rho)$, where σ_{Π_ρ} is the optimal linear schedule for algorithm $(H(\bar{\rho}), D_\rho)$ and can be found by Procedure 3.1. Karp, et al., [1] have shown that $k(H(\bar{\rho}), D_\rho)$, where $\bar{\rho} \in J$, is bounded above by a constant. However, as they pointed out, to find the constant is still open. In [12], 25 algorithms were studied and it was found that the difference $k(J, D)$ is equal to either one or zero for those algorithms.

The following theorem provides conditions on the dependence matrix of a given algorithm which guarantee that an optimal linear schedule for a given computation yields the same execution time as the free schedule.

THEOREM 4.1. Without the loss of generality, let $D_\rho = [\bar{d}_1, \ldots, \bar{d}_{m_\rho}]$. If $rank\{[\bar{d}_1 - \bar{d}_2, \bar{d}_1 - \bar{d}_3, \ldots, \bar{d}_1 - \bar{d}_{m_\rho}]\} < rank(D_\rho)$, then $k((H(\bar{\rho}), D_\rho) = 0$.

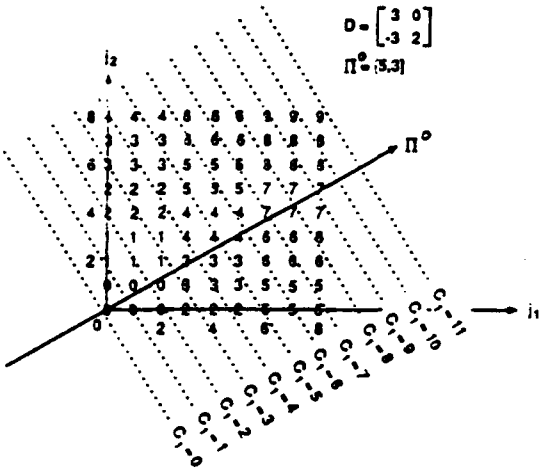
Proof. Provided in Appendix.

Theorem 4.1 applies to Example 3.2 where $k(H(\bar{\rho}), D_\rho) = 0$, $\bar{\rho} = [11, 0]^T$. Next, it is shown in two examples that, $k(J, D)$, the difference between execution times of algorithm (J, D) achieved by optimal linear schedules and free schedules, depends on the shape and size of the index set of the algorithm. In other words, the result of Theorem 4.1 does not generalize to the case when all computations are considered.

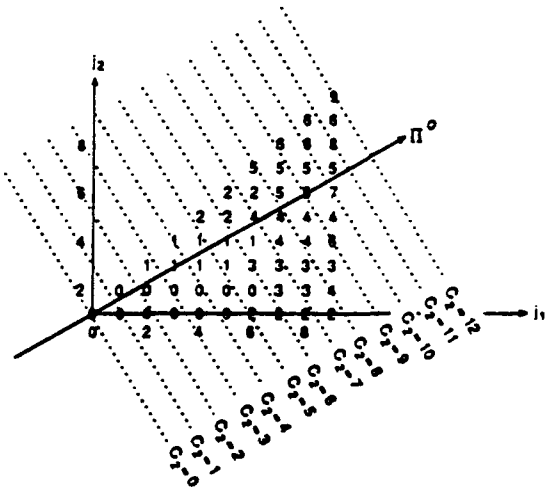
Example 4.1. Consider algorithm (J, D) where $J = \{[j_1, j_2]^T: 0 \le j_1, j_2 \le s, s, j_1, j_2 \in Z\}$ and

$$D = \begin{bmatrix} 3 & 0 \\ -3 & 2 \end{bmatrix}.$$

Let Π^o be the linear schedule vector by which the total execution time of the whole algorithm is minimized. By Procedure 3.1 in [14], $\Pi^o = [5, 3]$. Figure 3a depicts the index set where $s = 8$. The label at each index point $\bar{j}$ is equal to $\sigma_f(\bar{j})$, and the parallel lines correspond to the hyperplanes $\sigma_{\Pi^o}(\bar{j}) = \lfloor([5, 3]\bar{j})/6\rfloor = c_1$. If an index point $\bar{j}$ lies on the line $\sigma_{\Pi^o}(\bar{j}) = c_1$ or between lines $\sigma_{\Pi^o}(\bar{j}) = c_1$ and $\sigma_{\Pi^o}(\bar{j}) = c_1 + 1$, then the computation indexed by $\bar{j}$ is assigned to execute at time step c_1 by the optimal linear schedule σ_{Π^o}. It can be verified that when $s = 6$, $k(J, D) = 0$ and when $s = 8$, $k(J, D) = 10 - 9 = 1$. So $k(J, D)$ is a function of the size s. Actually, regardless of the value of s, $k(J, D)$ is bounded above by constant $\lfloor(\Pi^o[2, 0]^T)/6\rfloor = 1$. The derivation of this upper bound can be done heuristically as follows. Clearly, the last computation scheduled to execute by the optimal linear schedule σ_{Π^o} is indexed by the index point $\bar{\rho} = [s, s]^T$. Consider $(H(\bar{\rho}), D_\rho)$; because D_ρ satisfies the condition in Theorem 4.1, $k(H(\bar{\rho}), D_\rho) = 0$. Also, $\Pi_\rho = \Pi^o$ due to the fact that $D_\rho = D$. So, $k(J, D) = \sigma_{\Pi^o}(\bar{u})$ where $\bar{u}$ is an initial point ($\bar{u}$ is an initial point if $\sigma_f(\bar{u}) = 0$) such that $\sigma_{\Pi^o}(\bar{u}) = \min\{\sigma_{\Pi^o}(\bar{u}_i) = 0, \sigma_f(\bar{u}) = 0, \bar{u}_i \in H(\bar{\rho})\}$. There are six initial points in J. If the initial point $\bar{u}$ is not easy to identify, one possible upper bound for $k(J, D)$ is $\sigma_{\Pi^o}([2, 1]^T) = \max\{\sigma_{\Pi^o}(\bar{u}_i): \sigma_f(\bar{u}_i) = 0, \bar{u}_i \in J\} \ge \sigma_{\Pi^o}(\bar{u})$. As shown in [17], there is only one initial point connected to $\bar{\rho}$, and, regardless of the value of s, initial point $[2, 1]^T$ does not belong to $H(\bar{\rho})$ (due to the fact $\Pi^o[2, 1]^T \not\equiv \Pi^o\bar{\rho} (mod 6)$). So, the upper bound $\sigma_{\Pi^o}([2, 1]^T)$ is not tight. It can be verified that the initial point $[2, 0]^T$ belongs to $H(\bar{\rho})$ (i.e., $\Pi^o[2, 0]^T = \Pi^o\bar{\rho} (mod 6)$) when s is such that 3 divides $4s - 2$ (i.e., $s = 2, 5, 8, 11$ and so on). Therefore, $k(J, D) \le \sigma_{\Pi^o}([2, 0]^T) = 1$ and when $s = 2 + 3t$, $t \in Z$, $k(J, D) = \sigma_{\Pi^o}([2, 0]^T) = 1$.



(a)



(b)

Fig. 3. The index sets of the algorithms of Example 4.1 (a) and Example 4.2 (b). The label at each point equals the time step of execution assigned by the free schedule. The parallel lines indicate the time step of execution assigned by the optimal linear schedule. All points lying on the line $\sigma_{\Pi^o}(\bar{j}) = c$ or between lines $\sigma_{\Pi^o}(\bar{j}) = c$ and $\sigma_{\Pi^o}(\bar{j}) = c+1$ are executed at time step c.

Example 4.2. Consider the algorithm that has the same dependence matrix as Example 4.1 and $J = \{\bar{j}: A\bar{j} \le \bar{b}, \bar{j} \in Z^2\}$ where

$$A = \begin{bmatrix} -1 & 1 \\ 0 & -1 \\ 1 & 0 \end{bmatrix} \text{ and } \bar{b} = \begin{bmatrix} 0 \\ 0 \\ s \end{bmatrix}.$$

The index set is shown in Figure 3b (for $s = 3$) and has a shape different from that of Example 4.1. By Procedure 3.1 in [14], $\Pi^o = [5, 3]$. The linear schedule specified by Π^o is $\sigma_{\Pi^o}(\vec{j}) = \lfloor([5, 3]\vec{j})/6\rfloor$. When $s = 6$, $k(J, D) = 3$ and when $s = 8$, $k(J, D) = 10 - 6 = 4$, i.e., for the same sizes considered in Example 4.1 $k(J, D)$ has different values due to the different index set shape. Regardless of the value of s, $k(J, D)$ is bounded above by $\lfloor(\Pi^o[5, 1]^T)/6\rfloor = 4$. The derivation of this upper bound is similar to the one presented in Example 4.1.

5. Concluding Remarks

In summary, this paper provides a solution to the problem of identifying an optimal linear schedule for the execution of one specific computation of algorithms with uniform dependences. Also, a class of algorithms for which the optimal linear schedule is as efficient as the best possible schedule (i.e., it is as fast as the free schedule) is identified.

6. Appendix

Proof of Theorem 3.1. First, $\vec{u}$ is an initial point if and only if $\sigma_\Pi(\vec{u}) = 0$. Let $\vec{u}_1, \ldots, \vec{u}_s$, be all the initial points in $H(\vec{\rho})$. Because the computation indexed by $\vec{\rho}$ is the last to execute and the computation indexed by one of the initial points in $H(\vec{\rho})$ is the first one to execute, Equation 2.4 can be rewritten as

$$\begin{cases} \min f_\rho = \dfrac{\max\{\Pi(\vec{\rho} - \vec{u}_i): i = 1, \ldots, s\}}{\min\{\Pi\,\vec{d}_i: \vec{d}_i \in D_\rho\}} \\ \textit{Subject to } \Pi D_\rho > \vec{0} \end{cases} \tag{6.1}$$

Let's consider the linear schedule vector Π defined in Definition 2.5, i.e., the linear schedule vector Π such that $\min\{\Pi\vec{d}_i: \vec{d}_i \in D_\rho\} = 1$. Then, Equation 6.1 is equivalent to

$$\begin{cases} \min f_\rho = \max\{\Pi(\vec{\rho} - \vec{u}_i): i = 1, \ldots, s\} \\ \textit{subject to } \Pi D_\rho \geq \vec{1} \end{cases} \tag{6.2}$$

Furthermore, let $\vec{v}_i = \vec{\rho} - \vec{u}_i$, $i = 1, \ldots, s$. Using the approach described in ([18], pp. 96), Equation 6.2 can be reformulated as a linear programming problem:

$$\begin{cases} \min f = \pi_{n+1} \\ \textit{subject to } \Pi D_\rho \geq \vec{1} \\ \qquad \Pi[\vec{v}_1, \ldots, \vec{v}_s] \leq \pi_{n+1} \end{cases} \tag{6.3}$$

or equivalently

$$\begin{cases} \max -f = [\pi_{n+1}, \pi_1, \ldots, \pi_n] \begin{bmatrix} -1 \\ \vec{0} \end{bmatrix} \\ \textit{subject to } [\pi_{n+1}, \pi_1, \ldots, \pi_n] \begin{bmatrix} \vec{0} & -1 & \ldots & -1 \\ -D_\rho & \vec{v}_1 & \ldots & \vec{v}_s \end{bmatrix} \\ \qquad \leq [-\vec{1}, \vec{0}] \end{cases} \tag{6.4}$$

Clearly, if $[\pi_{n+1}, \pi_1, \ldots, \pi_n]$ is an optimal solution to linear programming problem 6.4, then $[\pi_1, \ldots, \pi_n]$ is an optimal solution to (6.2), or Problem 2.1. The dual problem of Equation 6.4 is ([19], pp. 86):

$$\begin{cases} \min [-\vec{1}, \vec{0}]\vec{x} \\ \textit{subject to } \begin{bmatrix} \vec{0} & -1 & \ldots & -1 \\ -D_\rho & \vec{v}_1 & \ldots & \vec{v}_s \end{bmatrix} \vec{x} = \begin{bmatrix} -1 \\ \vec{0} \end{bmatrix} \\ \qquad \vec{x} \geq \vec{0} \end{cases} \tag{6.5}$$

where $\vec{x} \in I\!R^{(m'_\rho - s)\times 1}$. Now, Problem 2.1 has been formulated as linear programming problem 6.4. Next, the solution of linear programming problem 6.5 is considered first. The solution of linear programming problem 6.4 can be constructed easily from the solution of (6.5).

Let's assume that $m'_\rho = rank(D_\rho) = n$. Let $t(\Pi, H(\vec{\rho}), D_\rho)$ denote the total execution time achieved by σ_Π for $(H(\vec{\rho}), D_\rho)$, i.e., $t(\Pi, H(\vec{\rho}), D_\rho)$ is expressed by Equation 2.3. Consider the linear schedule vector $\Pi'' \in C^\rho$ such that $t(\Pi'', H(\vec{\rho}), D_\rho) = \min\{t(\Pi, H(\vec{\rho}), D_\rho): \Pi \in C^\rho\}$. Without loss of generality, let $\Pi'' = \Pi(1 \ldots n/1 \ldots n)$. Let $G = [\vec{d}_1, \ldots, \vec{d}_n]$, then $\Pi'' = \beta\vec{1}G^{-1}$. Clearly, if it can be shown that $\Pi^* = (1/\beta)\Pi'' = \vec{1}G^{-1}$ is an optimal solution of linear programming problem 6.2, we are done. Now let's assume that $\vec{u}_1 = \vec{\rho} - \vec{v}_1$ is an initial point to be executed first by Π^*, i.e., $\Pi^*\vec{u}_1 = min\{\Pi^*\vec{j}: \vec{j} \in H(\vec{\rho})\}$ and consider the following linear programming problem

$$\begin{cases} \min \Pi\vec{v}_1 \\ \textit{subject to } \Pi D_\rho \geq \vec{1} \end{cases} \tag{6.6}$$

and its duality

$$\begin{cases} \max\ \bar{1}\ \bar{y} \\ \text{subject to } D_o\bar{y} = \bar{v}_1 \\ \qquad\qquad \bar{y} \geq \bar{0} \end{cases} \tag{6.7}$$

where $\bar{y} \in I\!R^{m_o \times 1}$. By the definition of Π^*, it is an optimal solution of problem 6.6, i.e., G is an optimal basis for problem 6.6. Let $D_o = [G, H]$, $C_G = [-1, \ldots, -1]_{1\times n}$ and $C_H = [-1, \ldots, -1]_{1\times(m_o - n)}$. By the simplex procedure ([19], pp. 58) the initial tableau for problem 6.7 takes form

$$\begin{bmatrix} G & H & \bar{v}_1 \\ C_G & C_H & 0 \end{bmatrix}$$

If matrix G is used as a basis, then the corresponding tableau becomes

$$\begin{bmatrix} I & G^{-1}H & G^{-1}\bar{v}_1 \\ \bar{0} & C_H - C_G G^{-1}H & -C_G G^{-1}\bar{v}_1 \end{bmatrix}$$

where I is the identity matrix. Because Π^* is an optimal solution, by the *Optimality Condition Theorem* ([19], pp. 43), $G^{-1}\bar{v}_1 \geq \bar{0}$ and $C_H - C_G G^{-1}H \geq \bar{0}$.

For linear programming problem 6.5, the initial tableau takes form

$$\begin{bmatrix} \bar{0} & -1 & \ldots & -1 & -1 \\ -D_o & \bar{v}_1 & \ldots & \bar{v}_s & \bar{0} \\ -\bar{1} & 0 & \ldots & 0 & 0 \end{bmatrix}$$

Let

$$G_o = \begin{bmatrix} \bar{0} & -1 \\ -G & \bar{v}_1 \end{bmatrix},$$

$$K = \begin{bmatrix} \bar{0} & -1 & \ldots & -1 \\ -H & \bar{v}_2 & \ldots & \bar{v}_s \end{bmatrix},$$

$$C_{G_o} = [C_G, 0],\ \bar{w} = [-1, \bar{0}]^T_{1\times(n+1)}$$

and

$$C_K = [C_H, \bar{0}]_{1\times(m_o - n + s - 1)}.$$

If G_o is used as a basis for problem 6.5, the corresponding tableau for problem 6.5 is

$$\begin{bmatrix} I & G_o^{-1}K & G_o^{-1}\bar{w} \\ \bar{0} & C_K - C_{G_o}G_o^{-1}K & -C_G G_o^{-1}\bar{w} \end{bmatrix} \tag{6.8}$$

After some manipulation it can be seen that

$$G_o^{-1} = \begin{bmatrix} -G^{-1}\bar{v}_1 & -G^{-1} \\ -1 & \bar{0} \end{bmatrix}.$$

So,

$$C_{G_o}G_o^{-1}K = [C_G, 0]\begin{bmatrix} G^{-1}\bar{v}_1 & G^{-1} \\ 1 & \bar{0} \end{bmatrix}\begin{bmatrix} \bar{0} & 1 & \ldots & 1 \\ H & -\bar{v}_2 & \ldots & -\bar{v}_s \end{bmatrix}$$

$$= -[\Pi^*\bar{v}_1, \Pi^*]\begin{bmatrix} \bar{0} & 1 & \ldots & 1 \\ H & -\bar{v}_2 & \ldots & -\bar{v}_s \end{bmatrix}$$

$$= [C_G G^{-1}H, \Pi^*(\bar{v}_2 - \bar{v}_1), \ldots, \Pi^*(\bar{v}_s - \bar{v}_1)]$$

and

$$C_K - C_{G_o}G_o^{-1}K = [C_H - C_G G^{-1}H, \Pi^*(\bar{v}_1 - \bar{v}_2), \ldots, \Pi^*(\bar{v}_1 - \bar{v}_s)]$$

$\Pi^*\bar{u}_1 = \min\{\Pi^*\bar{u}_i : i = 1, \ldots, s\}$ implies that $\Pi^*(\bar{v}_1 - \bar{v}_i) = \Pi^*(\bar{u}_i - \bar{u}_1) \geq 0$, $i = 1, \ldots, s$. Therefore, $C_K - C_{G_o}G_o^{-1}K \geq \bar{0}$. For $G_o^{-1}\bar{w}$, because $G^{-1}\bar{v}_1 \geq \bar{0}$, it follows

$$G_o^{-1}\bar{w} = \begin{bmatrix} G^{-1}\bar{v}_1 & G^{-1} \\ 1 & \bar{0} \end{bmatrix}\begin{bmatrix} 1 \\ \bar{0} \end{bmatrix} = \begin{bmatrix} G^{-1}\bar{v}_1 \\ 1 \end{bmatrix} \geq \bar{0}$$

Now it has been shown that $C_K - C_{G_o}G_o^{-1}K \geq \bar{0}$ and $G_o^{-1}\bar{w} \geq \bar{0}$ for tableau 6.8. Again by the *Optimality Conditions Theorem* ([19], pp. 43), G_o is an optimal basis for problem 6.5.

According to the theorem ([19], pp. 91), if G_o is an optimal basis for problem 6.5, then $\Pi' = C_{G_o}G_o^{-1} = -C_G[G^{-1}\bar{v}_1, G^{-1}] = [\Pi^*\bar{v}_1, \Pi^*]$ is an optimal solution for problem 6.4. Therefore, the optimal solution of problem 6.2 is $\Pi_o = \Pi^*$ and the optimal execution time is $\pi_{n+1} = \Pi^*\bar{v}_1$. So the optimal solution of Problem 2.1 for $(H(\bar{\rho}), D_o)$ is $\Pi(1\ldots n/1\ldots n)$ which belongs to C^o, i.e., $\Pi_o \in C^o$.

Now let's discuss the case where $rank(D_o) = m'_o < n$. $H(\bar{\rho})$ can be transformed into an m'_o-*dimensional* space by a bijective mapping. Without loss of

generality, let $\bar{d}_1, \ldots, \bar{d}_{m'_\rho}$ be linearly independent and $D'_\rho = [\bar{d}_1, \ldots, \bar{d}_{m'_\rho}]$. Three facts are proven next. First, let $T \in Z^{m'_\rho \times n}$ and $TH(\bar{\rho})$ denote the set $\{T\bar{j}: \bar{j} \in H(\bar{\rho})\}$. Then the mapping τ: $H(\bar{\rho}) \rightarrow TH(\bar{\rho})$, $\tau(\bar{j}) = T\bar{j}$, $\bar{j} \in H(\bar{\rho})$, is bijective if and only $rank(TD'_\rho) = m'_\rho$. Because $rank(D_\rho) = m'_\rho$, $\bar{d}_i$ can be written as a linear combination of $\bar{d}_1, \ldots, \bar{d}_{m'_\rho}$, $i = m'_\rho + 1, \ldots, m_\rho$, i.e., there exists a matrix $\Gamma \in I\!R^{m'_\rho \times (m_\rho - m'_\rho)}$ such that $D_\rho = D'_\rho[I, \Gamma]$ where I is the identity matrix. If $\bar{j}_1, \bar{j}_2 \in H(\bar{\rho})$, then $\bar{j}_1 = \bar{j}_2 + D_\rho\bar{\lambda}$ where $\bar{\lambda} \in Z^{m_\rho}$. So, $T(\bar{j}_1 - \bar{j}_2) = TD_\rho\bar{\lambda} = TD'_\rho[I, \Gamma]\bar{\lambda} = TD'_\rho\,\bar{\alpha}$ where $\bar{\alpha} = [I, \Gamma]\bar{\lambda}$. If $rank(TD'_\rho) = m'_\rho$, then equation $TD'_\rho\bar{\alpha} = \bar{0}$ has no nontrivial solution which means that $T(\bar{j}_1 - \bar{j}_2) = \bar{0}$ if and only if $\bar{j}_1 - \bar{j}_2 = \bar{0}$. Therefore, τ is bijective.

Secondly, there always exists such matrix T that $rank(TD'_\rho) = m'_\rho$. Because $rank(D'_\rho) = m'_\rho$, there exists a set of m'_ρ linearly independent rows of D'_ρ, i.e., there exists a nonsingular submatrix $D(1 \ldots m'_\rho / r_1 \ldots r_{m'_\rho})$, $1 \le r_1 < \ldots < r_{m'_\rho} \le n$. let

$$T = \begin{bmatrix} E_{r_1} \\ \ldots \\ E_{r_{m'_\rho}} \end{bmatrix},$$

then $TD'_\rho = D(1 \ldots m'_\rho / r_1 \ldots r_{m'_\rho})$ and $rank(TD'_\rho) = m'_\rho$.

Thirdly, $\Pi_\rho = \Pi(c_1 \ldots c_{m'_\rho} / r_1 \ldots r_{m'_\rho})$. Consider the algorithm $(TH(\bar{\rho}), TD_\rho)$ where $rank(TD_\rho) = m'_\rho$. If the same reasoning in the case where $m'_\rho = n$ is applied, then $\Delta = [D^{-1}(c_1 \ldots c_{m'_\rho} / r_1 \ldots r_{m'_\rho})$ is the optimal solution of Problem 2.1 for $(TH(\bar{\rho}), TD_\rho)$. If Δ is an optimal solution for $(TH(\bar{\rho}), TD_\rho)$, then ΔT is an optimal solution of Problem 2.1 for $(H(\bar{\rho}), D_\rho)$. This is true because

$$\max\{\Delta(T\bar{\rho} - T\bar{u}_i): i = 1, \ldots, s\} = \max\{\Delta T(\bar{\rho} - \bar{u}_i): i = 1, \ldots, s\} \qquad (6.9)$$

The left-hand side of Equation 6.9 is the optimal value of the objective function in Equation 6.2 achieved by Δ for $(TH(\bar{\rho}), TD_\rho)$. So the right-hand side of Equation 6.9 should be optimal for $(H(\bar{\rho}), D_\rho)$ and ΔT should be an optimal linear schedule vector for $(H(\bar{\rho}), D_\rho)$. Therefore, the optimal solution of Problem 2.1 for $(H(\bar{\rho}), D_\rho)$ is $\Pi_\rho = \Delta T = \Pi(c_1 \ldots c_{m'_\rho} / r_1 \ldots r_{m'_\rho})$ which belongs to C^ρ, i.e., $\Pi_\rho \in C^\rho$.

Proof of Theorem 4.1. First, it is shown that there exists a linear schedule vector Π such that $\Pi\bar{d}_1 = \ldots = \Pi\bar{d}_{m_\rho} \neq 0$. Let $H = [\bar{d}_1 - \bar{d}_2, \bar{d}_1 - \bar{d}_3, \ldots, \bar{d}_1 - \bar{d}_{m_\rho}]$, consider the following two equations:

$$\text{(a) } \Pi H = \bar{0} \quad \text{and} \quad \text{(b) } \Pi D_\rho = \bar{0} \qquad (6.10)$$

Let Ω_1 and Ω_2 denote the solution space of Equation 6.10a and the solution space of Equation 6.10b, respectively. Clearly, if $\Pi \in \Omega_2$, then $\Pi \in \Omega_1$, so, $\Omega_2 \subseteq \Omega_1$. Let $dim\{\Omega_i\}$, $i = 1, 2$, denotes the number of linearly independent vectors in Ω_i. Because $rank(H) < rank(D_\rho)$, $dim\{\Omega_1\} = n - rank(H) > n - rank(D_\rho) = dim\{\Omega_2\}$. This implies that $\Omega_2 \subset \Omega_1$. So there exists at least one vector Π belonging to Ω_1 but not belonging to Ω_2, i.e., $\Pi\bar{d}_1 = \ldots = \Pi\bar{d}_{m_\rho} \neq 0$.

The optimal linear schedule vector Π_ρ for $(H(\bar{\rho}), D_\rho)$ can be selected as follows. If $\Pi\bar{d}_1 = \ldots = \Pi\bar{d}_{m_\rho} < 0$, the $\Pi_\rho = -\Pi$, otherwise $\Pi_\rho = \Pi$. Now,

$$t(\Pi_\rho, H(\bar{\rho}), D_\rho) = \left\lceil \frac{\Pi_\rho\bar{\rho} - \min\{\Pi_\rho\bar{j}: \bar{j} \in H(\bar{\rho})\} + 1}{disp\ \Pi_\rho} \right\rceil$$

Let $\bar{g}$ be such that $\Pi_\rho\bar{g} = \min\{\Pi_\rho\bar{j}: \bar{j} \in H(\bar{\rho})\}$. Because $\bar{g} \in H(\bar{\rho})$, there exists a vector $\bar{\lambda} \in Z^{m_\rho}$ such that $\bar{\rho} = D_\rho\bar{\lambda} + \bar{g}$. The length of the path connecting $\bar{g}$ to $\bar{\rho}$ is equal to

$$l = \sum_{i=1}^{m_\rho} \lambda_i.$$

So $\Pi_\rho\bar{\rho} - \Pi_\rho\bar{g} = \Pi_\rho D_\rho\bar{\lambda} = disp\ \Pi_\rho \sum_{i=1}^{m_\rho} \lambda_i = l \cdot disp\ \Pi_\rho$.

Therefore,

$$t(\Pi_\rho, H(\bar{\rho}), D_\rho) = \left\lceil \frac{\Pi_\rho\bar{\rho} - \Pi_\rho\bar{g} + 1}{disp\ \Pi_\rho} \right\rceil = \left\lceil \frac{l\ disp\ \Pi_\rho + 1}{disp\ \Pi_\rho} \right\rceil = l + 1$$

Now, $t(\sigma_f, H(\bar{\rho}), D_\rho) \le t(\Pi_\rho, H(\bar{\rho}), D_\rho) = l + 1$. However, $t(\sigma_f, H(\bar{\rho}), D_\rho) \ge l + 1$ because there exists a path with length $l + 1$. So $t(\sigma_f, H(\bar{\rho}), D_\rho) = l + 1 = t(\Pi_\rho, H(\bar{\rho}), D_\rho)$, i.e. $k(H(\bar{\rho}), D_\rho) = 0$.

List of Symbols

- C^ρ: candidate set for $(H(\bar{\rho}), D_\rho)$; see Theorem 3.1.
- D: dependence matrix with n rows and m columns; see Definition 2.1 (4).
- D_ρ: matrix of dependencies that are present in $H(\bar{\rho})$; see the paragraph before Problem 2.1.
- $\bar{d}_i$: dependence (column) vector with n components; see Definition 2.1 (4).
- E: the set of edges of the algorithm precedence graph; see Definition 2.2

E_i: row vector with n components whose entries are zero except that ith entry is 1.

f_p: objective function of Problem 2.1.

$H(\bar{\rho})$: set of index point $\bar{\rho}$ and all its predecessors; see the paragraph before Problem 2.1.

I: identity matrix.

IR: set of real numbers.

J: index set; see Definition 2.1 (1).

$\bar{j}$: column vector; index point; see Definition 2.1 (1).

$k(J, D)$: The difference between the total execution times of (J, D) achieved by the optimal linear schedule and the free schedule, respectively; see second paragraph of Section 4.

m: number of dependence vectors in D; see Definition 2.1 (4).

m': number of dependence vectors in D_p.

m'_p: rank of matrix D_p.

N: set of non-negative integers.

N^+: set of positive integers.

n: number of components of index points in J; see Definition 2.1 (1).

Q: set of rational numbers.

$rank(A)$: rank of matrix A.

$t(\Pi, J, D)$: total execution time of (J, D) by σ_Π; see second paragraph of Section 4.

$t(\sigma_f, J, D)$: total execution time of (J, D) by σ_f; see second paragraph of Section 4.

$\bar{\rho}$: a specific index point in J; see Problem 2.1.

Z: set of integers.

Π: row vector; linear schedule vector; see Definition 2.6.

Π_p: row vector; optimal solution of Problem 2.1.

σ: a schedule (function); see Definition 2.3.

σ_f: free schedule; see Definition 2.4.

σ_Π: linear schedule; see Definition 2.6.

$\emptyset$: empty set.

$\bar{1}$: a column or row vector whose entries are all 1.

$\bar{0}$: a column or row vector whose entries are all 0.

References

1. R.M. Karp, R.E. Miller and S. Winograd. The organization of computations for uniform recurrence equations. *JACM* 14, 3, Jul. 1967, pp. 563–590.
2. D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Computers*, Vol. C-35, No. 1, Jan. 1986, pp. 1–12.
3. P.R. Cappello and K. Steiglitz. Unifying VLSI array designs with geometric transformations. *Proc. Int'l Conf. on Parallel Processing*, 1983, pp. 448–457.
4. P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. *Proc. 11th Annual Symposium on Computer Architecture*, 1984, pp. 208–214.
5. S.K. Rao. *Regular iterative algorithms and their implementations on processor arrays*. Ph.D. Dissertation, Stanford University, Stanford, California, Oct. 1985.
6. M. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, Dec. 1986, pp. 461–491.
7. J.-M. Delosme and I.C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. *Proc. Second Int'l Symposium on VLSI Technology, Systems and Applications*, 1985, pp. 268–273.
8. S.Y. Kung. *VLSI Array Processors*. Englewood Cliffs, N.J.: Prentice-Hall, 1987.
9. C. Guerra and R. Melhem. Synthesizing non-uniform systolic designs. *Proc. Int'l Conf. on Parallel Processing*, 1986, pp. 765–771.
10. G.-J. Li and B.W. Wah. The design of optimal systolic arrays. *IEEE Trans. Computers*, Vol. C-34, Jan. 1985, pp. 66–77.
11. M.T. O'Keefe and J.A.B. Fortes. A comparative study of two systematic design methodologies for systolic arrays. *Proc. Int'l Conf. on Parallel Processing*, 1986, pp. 672–675.
12. J.A.B. Fortes, F. Parisi-Presicce. Optimal linear schedule for the parallel execution of algorithms. *Proc. Int'l Conf. on Parallel Processing*, 1984, pp. 322–328.
13. R. Cytron. Doacross: Beyond vectorization for multiprocessors (extended abstract). *Proc. Int'l Conf. on Parallel Processing*, 1986, pp. 836–844.
14. W. Shang and J.A.B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *Proc. Int'l Conf. on Systolic Arrays*, May 1988, pp. 393–402.
15. D.A. Padua. *Multiprocessors: Discussion of theoretical and practical problems*. Ph.D. Thesis, Univ. of Illinois at Urb.-Champ., Rept. No. UIUCDCS-R-79-990, Nov. 1979.
16. J.-K. Peir and R. Cytron. Minimum distance: a method for partitioning recurrences for multiprocessors. *Proc. Int'l Conf. on Parallel Processing*, 1987, pp. 217–225.
17. W. Shang and J.A.B. Fortes. Independent partitioning of algorithms with uniform dependencies. *Proc. Int'l Conf. on Parallel Processing*, Vol. 2, 1988, pp 26–33.
18. P.E. Gill, W. Murray and M.H. Wright. *Practical Optimization*. New York: Academic Press, 1981.
19. D.G. Luenberger. *Linear and Nonlinear Programming*. Second Edition, Menlo Park, California: Addison-Wesley Publishing Company, 1984.

Weijia Shang received the B.S. degree in computer engineering and science from Changsha Institute of Technology in 1982 and the M.S. degree in electrical engineering from Purdue University in 1984.

She is currently a Ph.D. student in the School of Electrical Engineering, Purdue University, West Lafayette, IN. Her research interests include parallel processing, computer architecture, scheduling problems, and systolic arrays.

Ms. Shang is a student member of IEEE and ACM.

Jose A.B. Fortes has been with the faculty of Purdue University's School of Electrical Engineering since 1984.

His current interests in parallel processing include the systematic design of algorithmically specialized processor-array architectures, automatic parallelism detection and exploitation techniques, and fault-tolerant computing.

Fortes has published over 40 technical papers in journals and conference proceedings in the areas of parallel processing, fault-tolerant computing, and VLSI architectures. He has worked on several projects in these areas in cooperation with or with funding from NSF, ONR, AT&T, GE, RCA, and NCR.

Fortes is a member of IEEE.

He received his M.S.E.E. and Ph.D. E.E. degrees from Colorado State University and the University of Southern California in 1981 and 1983, respectively.

REFERENCE NO. 2

Chean, M., and Fortes, J. A. B., "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays," *IEEE Computer,* December 1990, pp. 55-69.

Note- This paper surveys and classifies processor array reconfiguration schemes proposed by the authors and other researchers. Criteria for evaluation of the schemes and their relative merits are also discussed.

SURVEY & TUTORIAL SERIES

A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays

Mengly Chean* and Jose A.B. Fortes
Purdue University

Very large scale integration/wafer-scale integration (VLSI/WSI) technology is most advantageous when used to implement regularly structured systems such as large arrays of identical processing elements. As integration levels increase and the sizes of arrays grow larger, the possibility of single or multiple faults occurring in a VLSI/WSI array heightens. These faults can occur during the operational lifetime of the array, as well as during its manufacturing process. In a non-fault-tolerant array structure, the failure of a single element can cause the array performance to degrade severely if not fatally.

On the other hand, the array might be able to operate in a fault-tolerant reconfigurable structure, even if a certain number of faults are present. This can be done by using a reconfiguration technique for restructuring the array statically (at fabrication time, for yield enhancement) or dynamically (during its operational lifetime, for improved reliability).

In recent years, a large number of reconfiguration schemes have been proposed. Before deciding which techniques are best suited for particular designs, it is important to understand their similarities and differences. This article introduces a taxonomy for reconfiguration techniques for fault-tolerant processor arrays (FTPAs) and discusses their distinguishing characteristics.

This proposed taxonomy focuses on FTPA reconfiguration techniques useful for comparing many possible schemes. It can help engineers and researchers understand existing approaches and develop new ones.

The article focuses on the characterization and classification of a specific feature of FTPAs, that is, reconfiguration, as opposed to Abraham et al.,[1] which surveys and discusses different features of FTPAs. Schemes can be differentiated from one another according to the type of redundancy (time or hardware), allocation of redundancy (local or global), replacement unit (processor or a set of processors), switching domain (global or local), and switching implementation (switching element, bus, or network). These characteristics can be used as the basis for a taxonomy of reconfiguration techniques as illustrated by the classification trees shown in Figures 1 and 2.

A detailed discussion of every tree leaf as a separate and distinct class is neither practical nor possible in this article. Instead, the techniques in four major subtrees are grouped into larger classes that correspond to the following distinguishing characteristics:

- local redundancy in hardware redundant schemes,
- set switching in schemes with global hardware redundancy,
- processor switching in schemes with global hardware redundancy, and
- time redundancy.

* Chean is now with the Bellaire Research Center, Shell Development Company, Houston, Texas.

0018-9162/90/0100-0055$01.00 © 1990 IEEE

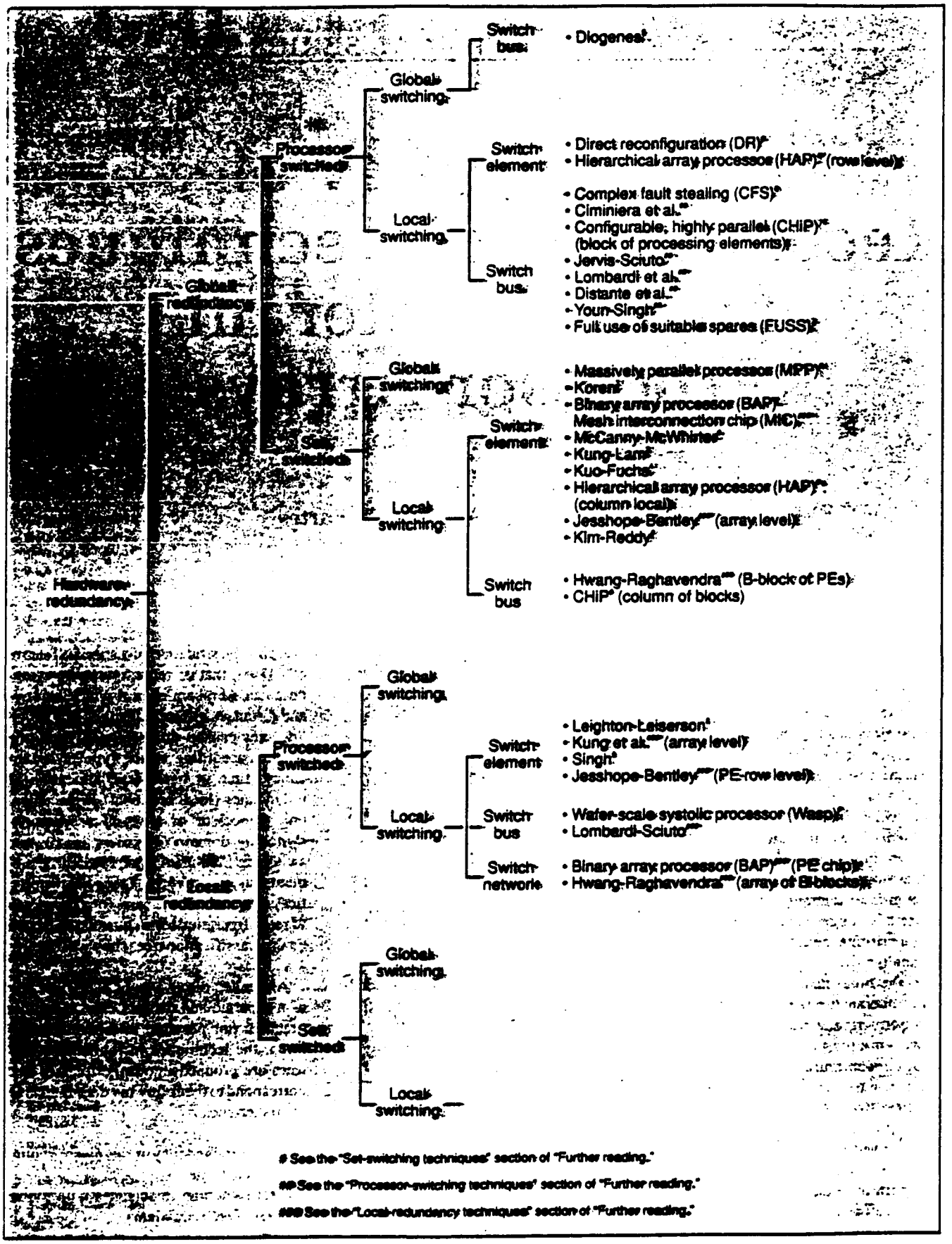


Figure 1. Taxonomy for hardware-redundancy reconfiguration schemes. (Asterisks identify the roots of the subtrees and their corresponding classes.)

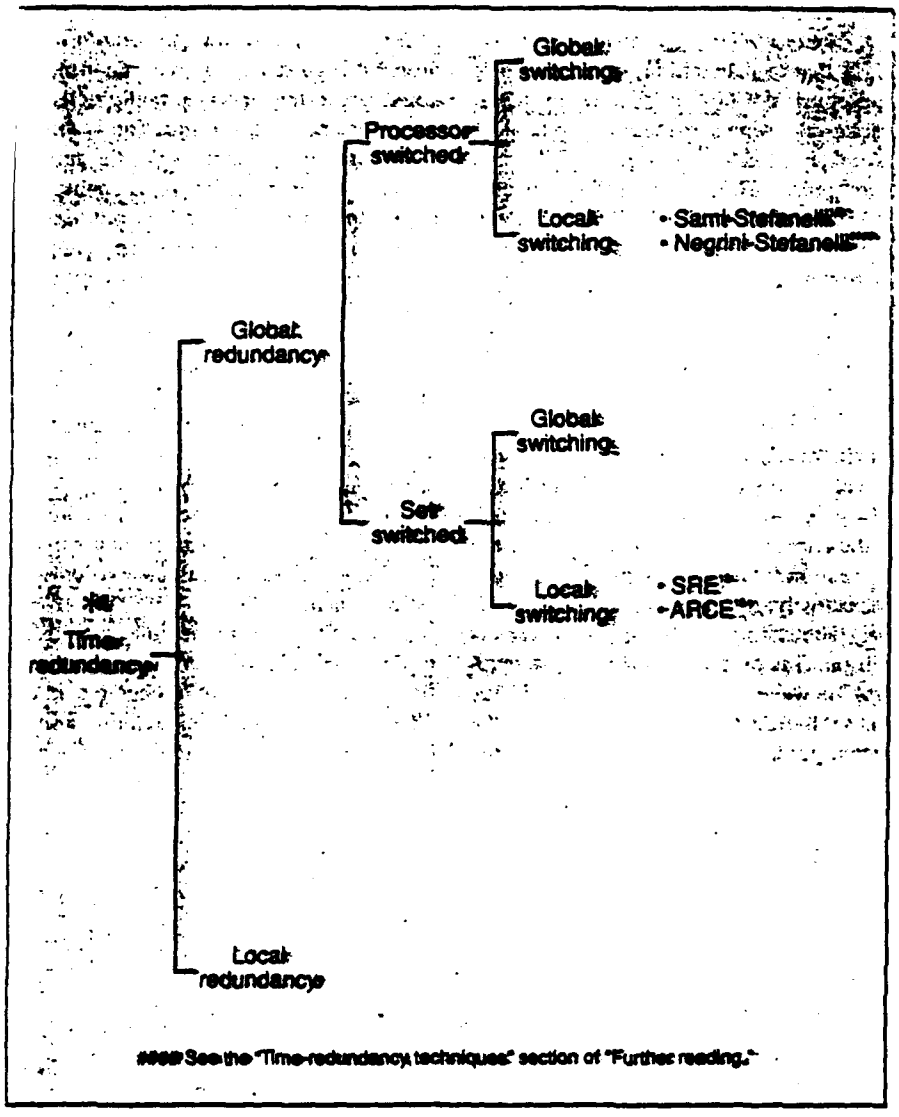


Figure 2. Taxonomy for time-redundancy reconfiguration schemes. (Asterisk identifies the root of the subtree and its corresponding class.)

In Figures 1 and 2, asterisks identify the roots of the subtrees to which these classes correspond. For brevity, these classes are denoted as the local-redundancy class, the processor-switching class, the set-switching class, and the time-redundancy class. Schemes in the local-redundancy class attempt to minimize the maximum wire length; this can result in some waste of functional cells. In processor-switching schemes, efficient cell use can be obtained at the cost of reconfiguration time and hardware. Reconfiguration algorithms in the set-switching class are less complex than local-redundancy approaches but can waste more functional cells. Finally, when a decrease in computational speed is an acceptable form of performance degradation, time-redundancy techniques can be considered and potentially have less silicon area overhead than those that use hardware redundancy.

This article is not designed to exhaustively survey previously proposed reconfiguration schemes. Instead, we review typical techniques selected from each class in the following sections to discuss and illustrate the main characteristics of different classes of schemes.

Terminology

The reconfiguration techniques presented in this article apply to an $(M+R) \times (N+C)$ array of cells in which M and N represent the number of rows and columns, respectively, and R and C represent the number of spare rows and spare columns, respectively. Figure 3b shows an array

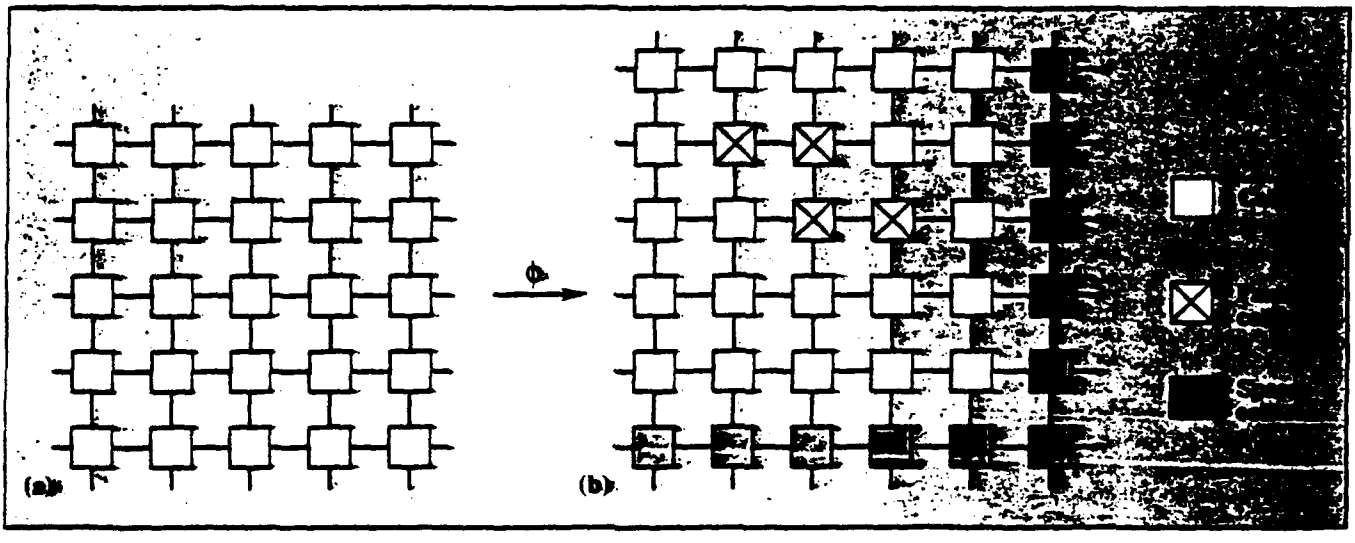


Figure 3. Mapping of a logical array into a physical array: (a) logical FTPA; (b) physical FTPA.

with one spare row and one spare column. Cells are denoted by squares, and crossed cells represent faulty cells. We will introduce additional circuitry used to bypass faulty cells when we discuss appropriate reconfiguration techniques.

A fault-tolerant processor array (see the sidebar, "The ubiquitous processor array," for background on processor arrays) is a two-dimensional array of identical and regularly interconnected processing elements (PEs) incorporating redundant circuitry (spares) and hardware for reconfiguration. A *cell* is a processing element of an FTPA. An *available cell* is a good, functional cell (also called fault-free or live cell). An *unavailable cell* refers to either a faulty cell or a fault-free cell that is already used to replace another cell.

A group of cells in a row, column, or block is called a *set*. A functional set contains cells that are all functional. In contrast, a set that contains one or more faulty cells is faulty. For example, a row that contains one or more faulty cells is a faulty row, and an array that contains one or more faulty cells is a faulty array.

A *physical FTPA* is a nonreconfigured FTPA that can contain faulty cells. A cell in a physical FTPA is denoted by (i, j), where i and j represent the cell coordinates, $1 \le i \le M+R$, $1 \le j \le N+C$. A *logical FTPA* is a reconfigured FTPA that is fully operational. A cell in a logical FTPA is denoted by $[i, j]$ where i and j represent its logical coordinates, $1 \le i \le M$, $1 \le j \le N$.

An array reconfiguration is a mapping ϕ of logical cells in a logical FTPA into functional cells in a physical FTPA (see Figure 3). Thus, $\phi([i, j]) = (k,l)$ if logical cell $[i, j]$ corresponds to physical cell (k,l). The inverse of ϕ maps a functional physical cell to a logical cell, for example, $\phi^{-1}((i, j)) = [m,n]$. Also, ϕ_r and ϕ_c map a logical cell to the corresponding physical row and the corresponding physical column, respectively; ϕ_r^{-1} and ϕ_c^{-1} are defined similarly. For example, $\phi_r([m,n]) = k$ if logical cell $[m,n]$ is on physical row k. If a mapping between all cells in a logical FTPA and cells in a physical FTPA exists, the reconfiguration is said to be successful.

The *probability of survival* $\sigma(x)$ (also called *survivability*) is the conditional probability of success in reconfiguring an array, given that it has x faulty cells. In some instances, reconfiguration is aimed at constructing an array with the largest possible number of functional cells; the target (logical) array size does not need to be fixed. In this case, the term *harvest* refers to the number of fault-free cells used to construct the largest logical array possible. Ideally, the harvest is the number of cells in the physical array minus the number of faulty cells. In an FTPA, if S is the total number of spare cells and F the total number of faulty cells, then the *spare demand* ρ is defined as $\rho = F/S$.

Four figures of merit, *simplicity*, *efficiency*, *area*, and *locality* (abbreviated SEAL), are useful to evaluate a reconfiguration scheme.

Simplicity refers to the time complexity of a reconfiguration algorithm as a function of the number of processors in an FTPA. A simple algorithm requires a short execution time that should be polyno-

The ubiquitous processor array

In its basic form, a processor array is simply a set of processors organized as a grid where each point corresponds to a processing element and lines between points correspond to communication links between processors. This mesh-like organization has been proposed since the early days of digital computers and has been rediscovered for many different applications and technologies. Brief discussions of the history of early processor array designs can be found in Uhr[1] and Preston and Duff.[2]

There are many variants of processor-array architectures. For example, processing elements can be as complex as a modern microprocessor or as simple as an arithmetic logic unit. In fact, memory arrays can be considered an extremely simple example of processor arrays, particularly in the case of associative memories. There are also many variations as to how instructions are executed (centralized versus distributed control) and how memory and input/output are organized. In addition, the basic mesh topology is often augmented by additional links or routing mechanisms to support arbitrary communication patterns. The references listed here provide many examples of these variants. Recently, a particular type of architecture, denoted systolic array, has been used in a large number of proposals for special-purpose architectures. A sample of this type of work can be found in *Computer*[3] and Mead and Conway.[4]

Processor arrays are pervasive parallel architectures for two main reasons: (1) they suit digital electronic technologies (particularly VLSI/WSI), and (2) they can be structurally well-matched to many applications. The ease of implementation results from its modularity and local connectivity. Modularity means that it suffices to design one module — a processing element and associated connections — that can then be replicated to build arrays of arbitrary size (up to a maximum determined by factors such as reliability, power, clock distribution). Local connectivity means that each processor communicates only to a few neighboring processors, thus causing little need for connection wires to crossover — a factor that facilitates planar VLSI/WSI implementation. Further reading on this subject is available in Mead and Conway.[4]

The mesh topology is well suited for many problems, including image processing and understanding, pattern recognition, adaptive array beamforming, matrix computations, partial differential equations, bit-level arithmetic, digital filtering, cellular automata, computer graphics, Monte Carlo simulations, and games. Nowadays, many special-purpose and/or commercial processor arrays are being used for these applications. Poor yield and reliability remain important obstacles to making these machines more integrated, less expensive, larger (in terms of numbers of processors), and more dependable.

References

1. Uhr, L., *Computer Arrays and Networks: Algorithm-Structured Parallel Architectures*, Academic Press, New York, 1982.
2. Preston, K., Jr., and M.J.B. Duff, *Modern Cellular Automata: Theory and Applications*, Plenum, New York, 1984.
3. *Computer* (special issue on systolic arrays), Vol. 20, No. 7, July 1987.
4. Mead, C., and L. Conway, *Introduction to VLSI Systems*, Section 8.3, "Algorithms for VLSI Processor Arrays," Addison-Wesley, Reading, Mass., 1980, pp. 271-292.

nially bounded.

Efficiency refers to spare use. An efficient scheme wastes none or very few spare cells and, thus, achieves a very high array survivability and harvest.

Area refers to the overhead of the added interconnect and reconfiguration circuitry. Low-overhead schemes are desirable because a large silicon area increases the probability of having more defective elements.

Locality means that physical interconnections between logically adjacent cells in a reconfigured array should have minimal lengths. It determines the maximum delay in signal propagation, therefore limiting the clock rate at which the array can operate.

Distinguishing between two types of array reconfiguration that differ in their goals and in when they take place is important. *Static reconfiguration* is performed at array fabrication time and often uses hard (fixed) interconnections to bypass defective cells to enhance yield. *Dynamic reconfiguration* is performed during the operational lifetime of the array and over time uses soft (programmable) interconnections to logically replace the faulty cells to improve array reliability.

The two types of reconfiguration also differ in the relative weighting of the SEAL figures of merit. For example, in dynamic reconfiguration, it is important to have simple schemes to avoid degradation of system performance. On the other hand, in static reconfiguration, using a somewhat slower technique to achieve higher efficiency can be desirable and affordable. In both types, however, a major consideration is to minimize area that can impact yield and reliability to different degrees. Finally, manufacturing defects and operational faults can have different distributions on the array, and evaluation of a particular scheme must take this fact into consideration.

Taxonomy for reconfiguration techniques

To illustrate reconfiguration characteristics, we use three schemes from Kung and Lam[2]:

• *Scheme I.* Faulty rows and/or columns are skipped (this is illustrated in Figure 4). A greedy algorithm can be used to first eliminate the row or column containing the most faulty cells.

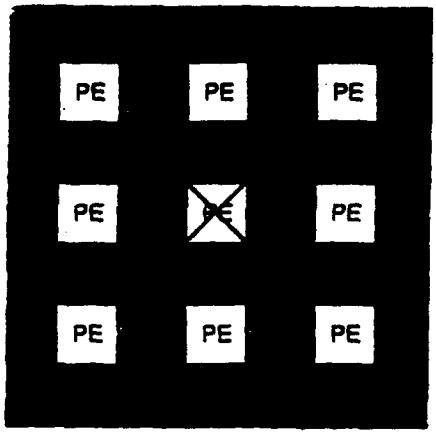


Figure 4. A (2 × 3) reconfigured FTPA (middle row is skipped over).

• *Scheme II.* The strategy is to build an array row by row, picking as the next cell the one that satisfies a predetermined criterion and that excludes the least number of live cells from being used. To accommodate delay between two adjacent cells in a logical array, some "delay registers" can be used. A simple "maze-runner" can be used to determine the interconnect requirements.

• *Scheme III.* The FTPA is partitioned into horizontal strips, and the cells in each strip are connected to form logical rows. These logical rows are then connected to form the whole array. Also, delay registers are used in the reconfigured array.

The taxonomy proposed in this article is based on a variety of reconfiguration characteristics, namely *type of redundancy*, *allocation of redundancy*, *replacement unit*, *switching domain*, and *switching mechanism*.

The type of redundancy can be hardware or time. Hardware redundancy indicates that spare cells are available to logically replace faulty cells, whereas time redundancy means that operational cells are assigned the computations that faulty cells would perform if they were operational. All three schemes outlined above use hardware redundancy. An array can use both types of redundancy (for example, when graceful degradation is allowed after all physical spares have been used), but this is equivalent to having two reconfiguration schemes for the same array.

The allocation of redundancy can be local or global. Local redundancy corresponds to the case when spares can logically replace only cells in specific parts of the array (horizontal strips in Scheme III above). In a broad sense, this is equivalent to partitioning the array into smaller blocks, each reconfigured locally. In some cases, when a block cannot be reconfigured, a block-replacement scheme can be used to substitute the full block for a good one. These techniques have a hierarchical nature and correspond to a combination of reconfiguration schemes that can belong to different classes. Global redundancy, on the other hand, means that any of the spare cells can logically replace any faulty cell. In some schemes, any cell can be either a spare or a regular cell.

The replacement unit can be a cell or a set of cells, and the array is said to be processor switched or set switched, respectively. In processor-switched arrays, the replacement of a faulty cell only requires the logical removal of that cell. On the contrary, in set-switched arrays, the replacement of a faulty cell requires the logical removal and replacement of a set of cells, such as a column or a row of cells (see Scheme I above).

The switching domain can be distributed or global. The form of connection between any two cells depends on their physical location when distributed (local) switching is used; topologically close cells are more easily connected, whereas distant cells cannot be connected or require more switches/buses/linking elements for the connection. In an array with global switching, the form of connection between any two cells is independent of their position in the array. Usually, distributed switching (as in all the schemes illustrated above) allows the interspersing of switching mechanisms with cells (for example, a switch lattice), whereas global switching requires the separation of the reconfiguration hardware from the processing elements (for example, a global bus or interconnection network).

The switching mechanism can be a switch element, a switch bus, or a switch network. A switch element consists of cells or other elements (for example, multiplexers or delay registers as in Schemes II and III) that are capable of routing data. In practice, this corresponds to the existence of bypassing mechanisms "inside" the elements that connect inputs directly to outputs. A switch bus consists of a bus and switches where different wires can be connected to different cells to establish a particular interconnection pattern. Distinct segments of each wire can also be

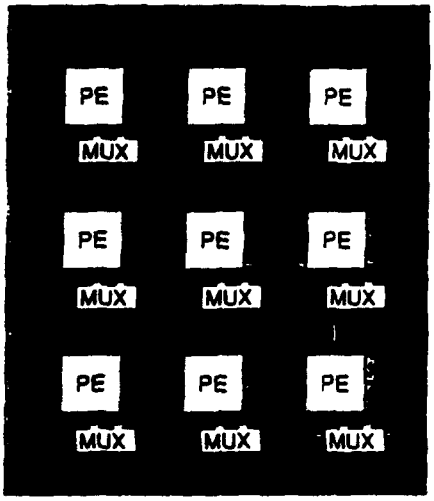


Figure 5. An FTPA structure with multiplexer hardware to support switching.

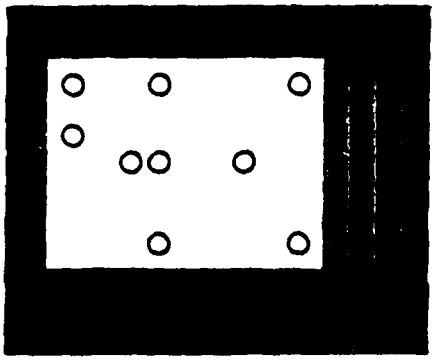


Figure 6. An example of the Kuo-Fuchs approach.[4]

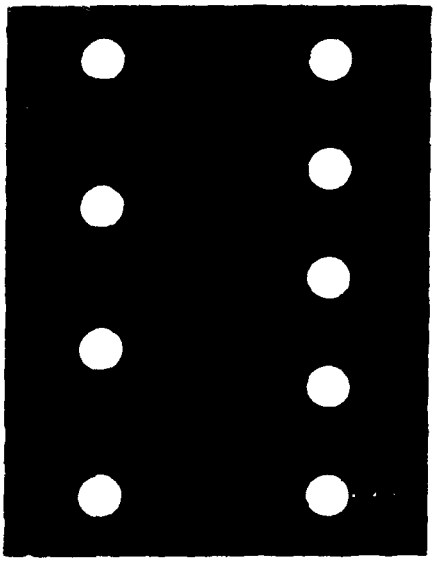


Figure 7. Bipartite description of the fault distribution in Figure 6.[4]

used to connect different pairs of cells, and switches are used to make or break connections between wire segments and between wires and cells. A switch network consists of a collection of interconnected switches that, depending on the setting of each switch, establish connections between the inputs and outputs of specific cells (for example, a crossbar network, a multistage interconnection network, or a lattice of switches interspersed among cells).

Figures 1 and 2 show the classification trees defined by the taxonomy just introduced. Each leaf of the trees corresponds to a different class of reconfiguration schemes. References for selected schemes discussed in this article (including the references in the "Further reading" sidebar) are shown next to the leaves of the trees. In these references, readers can find detailed descriptions of schemes in the classes specified by the respective leaves (however, readers are cautioned that some references describe several schemes, possibly from distinct classes). When the same scheme appears in more than one leaf, it indicates a hierarchical combination of two different schemes, each of which corresponds to a leaf or class. In these cases, in addition to a reference, the domain of reconfiguration is specified in parenthesis. As already mentioned, the large number of types of schemes determined by the taxonomy are grouped into four major classes. This is done to facilitate the discussion and comparison of fundamentally different approaches to the reconfiguration problem. Three of these four major classes correspond to hardware redundancy solutions, namely local redundancy, processor switching with global redundancy and set switching with global redundancy. The fourth major class contains schemes that use time redundancy. For simplicity, these major classes are called the local-redundancy class, the processor-switching class, the set-switching class, and the time-redundancy class. Asterisks indicate the roots of the subtrees (see Figures 1 and 2) that correspond to these classes.

Set-switching class

In the set-switching class, the replacement of a faulty cell requires the logical removal and replacement of a set of cells. Typical replacement sets are FTPA rows and columns. A typical array reconfiguration is shown in Figure 4 where one row of the array is skipped (assuming one spare row is provided). This is accomplished, for instance, by setting simple multiplexer switches that have been incorporated into the FTPA structure, as shown in Figure 5. The bypass circuitry (multiplexer) can be implemented inside a cell, as proposed in Koren.[3] Bypassing a faulty row is accomplished by turning all cells (including faulty cells) in that row into "connecting elements" to allow signals to go through them. The main advantage of the schemes in this class is their simplicity. Since the switches used to bypass (or to pass through) a faulty set are simple, the cost of redundant interconnect and control circuitry is minimal. On the other hand, the waste of cells is the main drawback.

There are several schemes in this class, many of which use a spare row, a spare column, or both. The main distinction among them is the way switch elements are implemented. Certain schemes also use multiple spare rows and/or columns, and a typical one is discussed next. (See "Further reading" for additional references on this topic.)

The Kuo-Fuchs strategy. Consider a (7+2) × (9+3) FTPA as shown in Figure 6, in which circles represent faulty cells. A general set-replacement algorithm can replace faulty rows/columns by proceeding from left to right and from top to bottom; that is, rows 1 and 3 can be replaced by the two spare rows, and columns 1, 3, and 4 can be replaced by the three spare columns. Obviously, this strategy fails to reconfigure the array. In the Kuo-Fuchs strategy,[4] rows and/or columns that contain the most number of faults are replaced first. To facilitate this, the FTPA is modeled as a bipartite graph whose two sets of nodes are array rows and columns that contain faulty cells, and edges are faulty cells. Figure 7 shows the bipartite description of the array in Figure 6. In the graph, R_i and C_j represent faulty row i and faulty column j, respectively. There are three edges from R_4, namely to columns C_3, C_4, and C_7, since there are three faulty cells (4,3), (4,4), (4,7). The Kuo-Fuchs approach replaces R_1 and R_4 with the two spare rows and C_1, C_4, and C_9 with the three spare columns, and the reconfiguration is successful.

The purpose of the bipartite description is to transform the reconfiguration problem into a graph-theoretic question, that is, that of finding the minimum cover of the bipartite graph mentioned above. Kuo and Fuchs[4] have shown that the optimal solution to the problem is NP-complete (nondeterministic polynomial-time complete)

and proposed two algorithms. One, for reasonably small arrays, uses a branch-and-bound approach to reduce the complexity of searching for a minimum cover. The other, for very large arrays, is a polynomial-time approximation algorithm that runs efficiently and often yields optimal solutions.

In summary, a scheme in the set-switching class is characterized by its replacement unit, whether it be a row, a column, or a subarray. Since the replacement unit is a set of cells, a very large number of spares might be required to provide sufficient fault tolerance for an array. However, the simplicity of the algorithm and the switching mechanism compensates for this wasteful use of spares. All set-switching techniques described here and in "Further reading" are summarized in Table 1, which shows the corresponding replacement sets.

Table 1. Summary of set-switching reconfiguration schemes.

Reconfiguration Scheme	Redundant Set
Massively parallel processor (MPP)#	Group of 128 × 4 processing elements
Binary array processor (BAP)-Mesh interconnection chip (MIC)###	Column of interconnect nodes
Koren[3]	Row/column of PEs
McCanny-McWhirter#	Row of PEs
Kim-Reddy#	Row of PEs
Kuo-Fuchs[4]	Row/Column of PEs
Hwang-Raghavendra###	Row of PEs
Configurable, highly parallel (CHiP)[10]	Column of blocks of processing elements
Jesshope-Bentley###	Row of PEs

See the "Set-switching techniques" section of "Further reading."

See the "Local-redundancy techniques" section of "Further reading."

Processor-switching class

In processor-switching schemes, an available cell directly or indirectly replaces a faulty cell. For this reason, the waste of functional cells can be reduced. There are two approaches in this class. In one approach, functional cells are systematically collected (such as in a linear fashion, as in Scheme II) to form a part (say, a row) of the target array. In another approach, faulty cells are replaced by their neighbors and those neighbors are further replaced by their neighbors and so on, in a chain fashion, until spare cells are reached. Next, we discuss the Diogenes scheme[5] as an example of the first approach, and the fault-stealing[6] and FUSS[7] schemes as examples of the second approach.

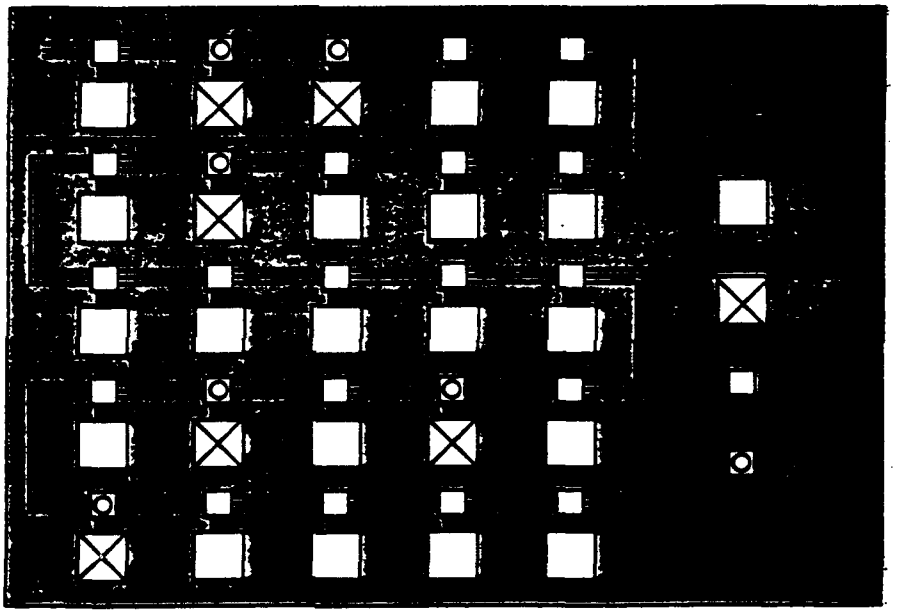


Figure 8. A possible layout for the Diogenes approach.

Diogenes strategy. In the Diogenes-strategy approach,[5] functional cells are collected by a mechanism consisting of programmable switches and bundles of wire that run above a line of cells and spare cells (that is, the array is linearized). Faulty cells are simply left unconnected. This procedure allows the Diogenes algorithm to achieve 100-percent use of fault-free PEs. Figure 8 illustrates the Diogenes concept. Small squares above individual cells represent switches that can connect fault-free cells (unmarked squares) or bypass faulty cells (squares marked with X). The switching emulates a stack (or queue) mechanism that "pushes" and "pops" wires "into" and "out of" processors. For an $N \times N$ FTPA (rectangular grid), the scheme needs three control lines and two bundles of wires, N lines for the stack and N (plus some other lines) for the queue.

In the Diogenes approach, the switch-bus mechanism is the most important part of the design. By adequately setting the switches, implementing different connection patterns is possible. For a specific desirable connectivity, the number of wires and switches can be chosen so that reconfiguring the array is always possible. One of the advantages of this approach includes the possibility of implementing different topologies, including two-dimensional arrays. Its stack/queue-like switch bus allows the scheme to achieve 100-percent use of fault-free cells. However, a large area must be allocated for the switch bus that might itself fail. Another potential disadvantage is that, in the presence of consecutive faulty PEs, logically adjacent PEs can be very far apart, thus requiring longer interconnect links. For this reason, additional vertical-shortcut paths across the array have been considered. The problem can be alleviated if somewhat simpler switching mechanisms are used at the expense of lower array survivabilities. This strategy is implemented in many schemes, two of which are

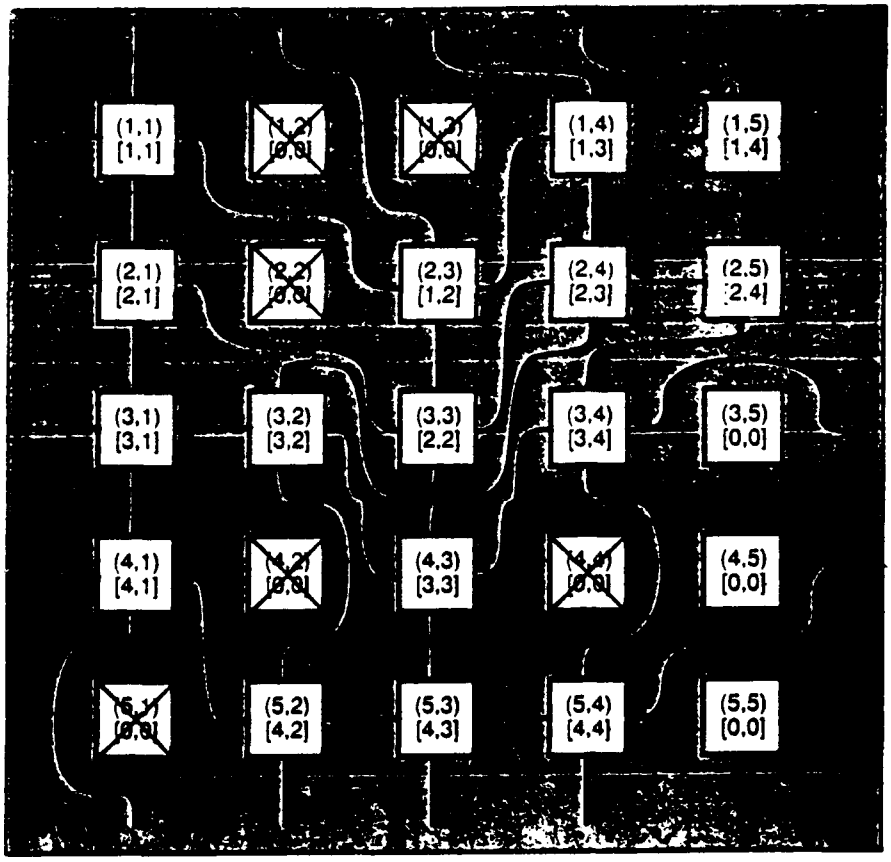


Figure 9. Array reconfigured by direct-reconfiguration (DR) scheme.

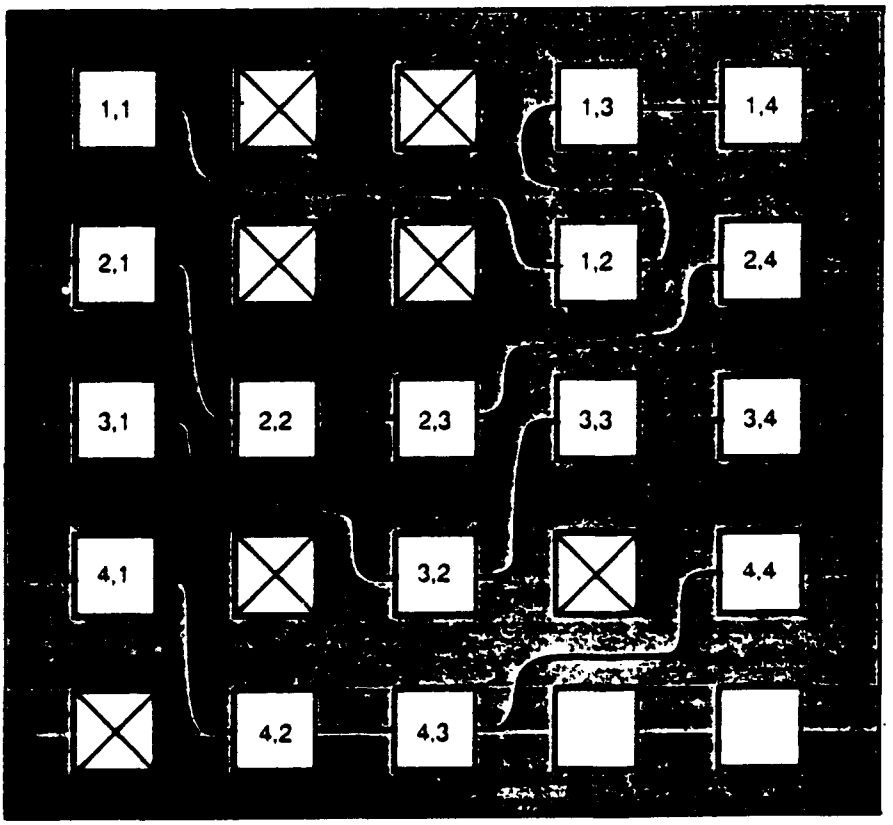


Figure 10. Array reconfigured by complex-fault-stealing (CFS) scheme.

presented next. The Diogenes approach is best for small arrays with large processing elements.

Fault-stealing strategy. Fault-stealing techniques,[6] also known as index-mapping schemes, reconfigure an FTPA in such a way that a faulty cell is replaced by a neighbor and this neighbor, in turn, is replaced by its neighbor and so on, until a spare cell is used. The spare cell is said to logically replace the faulty cell; the term "shift" is used to mean direct replacement; a cell is shifted to another if the former is directly replaced by the latter. For instance, in Figure 9, cell (1,3) can be replaced by (1,5) using the following successive direct replacements: $\phi([1, j]) = (1, j+1)$, $j = 3, 4$. This is referred to as a "chain shift."

Direct reconfiguration (DR) is a simple scheme that is the basis for the family of fault-stealing techniques. It uses one row and one column of spares ($R = C = 1$). The algorithm begins by scanning each column upward, marking the first faulty cell (if any) of each row as a vertical fault and other faults as horizontal faults. For the array shown in Figure 9, (5,1), (4,2), (1,3), and (4,4) are vertical faults, and other faulty cells are horizontal faults. Horizontal faults are chain-shifted right, and vertical faults are chain-shifted down. In Figure 9, (1,2) and (2,2) are chain-shifted right, and (4,2), (1,3), and (4,4) are chain-shifted down. Note that (1,2) and (2,2) were first shifted right and then shifted down (that is, they were shifted twice). The algorithm fails when there is more than one horizontal fault in one row of the array. Using this scheme, the array in Figure 9 is successfully reconfigured. The figure shows both physical and logical coordinates.

In DR, a faulty cell (i, j) can be shifted right to a fault-free cell $(i, j+1)$ or down to $(i+1, j)$. The set consisting of cells (i, j), $(i, j+1)$, and $(i+1, j)$ is referred to as an "adjacency domain." A family of schemes can be extended from the DR scheme by varying the size of (number of cells in) the adjacency domain and by modifying the replacement rules. The schemes in this family are called *fault-stealing schemes*. In order of increasing complexity, they are fault stealing of simple-fixed choice, simple-variable choice, complex-fixed choice, and complex-variable choice.

The *fixed-stealing algorithm* scans all rows in the order of increasing index numbers. In each row, the rightmost unavailable cell is shifted right, and other faulty cells "steal" (are vertically shifted

)) available cells on the next row. Variable stealing differs from fixed stealing in the choice of the unavailable cell in a row to shift right. Instead of selecting the rightmost unavailable cell, variable stealing selects the faulty cell that cannot steal a cell on the next row. The rest of the algorithm is basically the same as in DR.

The two algorithms we just described are referred to as simple-stealing algorithms. Vertically, (i, j) can be only shifted to $(i+1, j)$. In *complex fault stealing* (CFS), (i, j) can be also shifted to $(i+1, j+1)$. For an $(N+1) \times (N+1)$ array, the complete CFS algorithm is outlined as follows:

(1) Assume that in row i, $1 \le i \le N$, there are faulty or stolen (unavailable) cells (i,k_1), ... ,(i,k_s) with $k_1 < \ldots < k_s$.
(2) For each k_i, $0 < i < s$:
  (a) If $(i+1,k_i)$ is fault free, shift (i,k_i) to it.
  (b) Else if $(i+1,k_i+1)$ is fault free, shift (i,k_i) to it.
  (c) Otherwise, (i,k_i) is shifted right.
(3) If no cell is shifted along the row as per rule (2), then (i,k_s) is. Otherwise, (i,k_s) is shifted downward to either $(i+1,k_s)$ or $(i+1,k_s+1)$.

As an example, consider the array shown in Figure 10. In relation to the case shown in Figure 9, this array contains an additional faulty cell at (2,3). Were DR used, the reconfiguration would fail because there are two horizontal faults in the first row. However, if CFS is used, then the reconfiguration is successful, as Figure 10 shows. The figure shows logical coordinates and, to avoid cluttering, does not show connections for logical cells on the same columns.

The algorithm fails when, along a single row, two or more cells are shifted right, as in the case of Figure 11 where another fault occurs at cell (2,4). The interconnect links required by the algorithm become more complex than that required by DR, but can be implemented by the switch bus shown in Figure 12. On the other hand, the probability of survival is better than DR survivability.

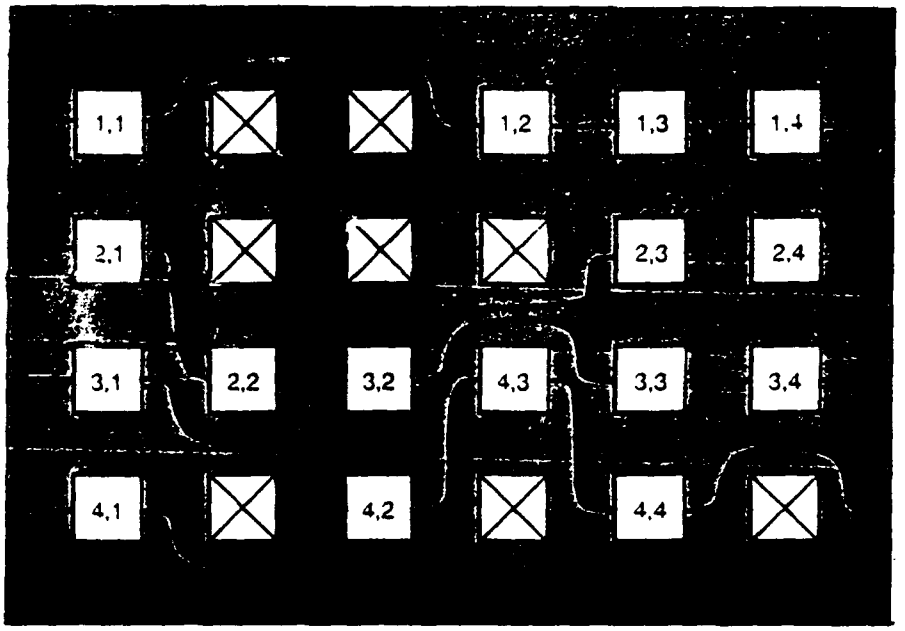


Figure 11. Array reconfigured by full-use-of-suitable-spares scheme with two columns of spares (FUSS-2).

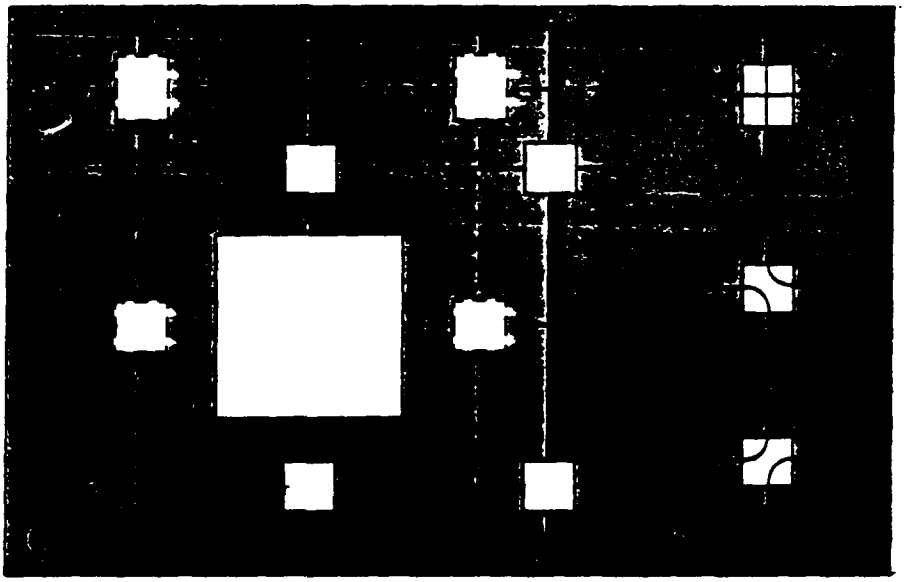


Figure 12. Switch-bus structure used in FUSS-1 and CFS: (a) cell and switches, and (b) switch settings.

FUSS strategy. A reconfiguration scheme called FUSS (Full Use of Suitable Spares)[7] uses an indicator vector called *a surplus vector* to guide the replacement of faulty cells in an FTPA. In its most general and ideal case, FUSS achieves 100-percent array survivability. Simple FUSS (or FUSS-C), a practical instance of FUSS, can achieve up to 99-percent probability of survival.

In FUSS-C, the FTPA is an $M \times (N+C)$ array in which C is the number of spare columns (spare rows are not used). Basically, the algorithm can be divided into four steps: *array preprocessing*, *surplus normalization*, *fault shifting*, and *cell interconnection*. This subdivision is done for clarity as well as for monitoring the algorithm progress.

In array preprocessing, the surplus vector is computed. Let f_i be the number of faulty cells in row i. The surplus vector is defined as

$$s = [s_1, s_2, \ldots, s_M]^T$$

where

$$s_i = \sum_{j=1}^{i} (C - f_j) = C \cdot i - \sum_{j=1}^{i} f_j$$

is the surplus of row i.

	1	2	3	4	5	6	s
1	0	1	1	0	0	0	0
2	0	1	1	1	0	0	-1
3	0	0	0	0	0	0	1
4	0	1	0	1	0	1	0

(a)

	1	2	3	4	5	6	s
1	0	1	1	0	0	0	0
2	0	1	1	1	0	0	0
3	0	3	0	2	0	0	0
4	0	1	0	1	0	1	0

(b)

1,1	0,0	0,0	1,2	1,3	1,4
2,1	0,0	0,0	0,0	2,3	2,4
3,1	2,2	3,3	4,3	3,3	3,4
4,1	0,0	4,2	0,0	4,4	0,0

(c)

Figure 13. An example of FUSS-1: (a) initial array A; (b) array B after shift; (c) final reconfigured array.

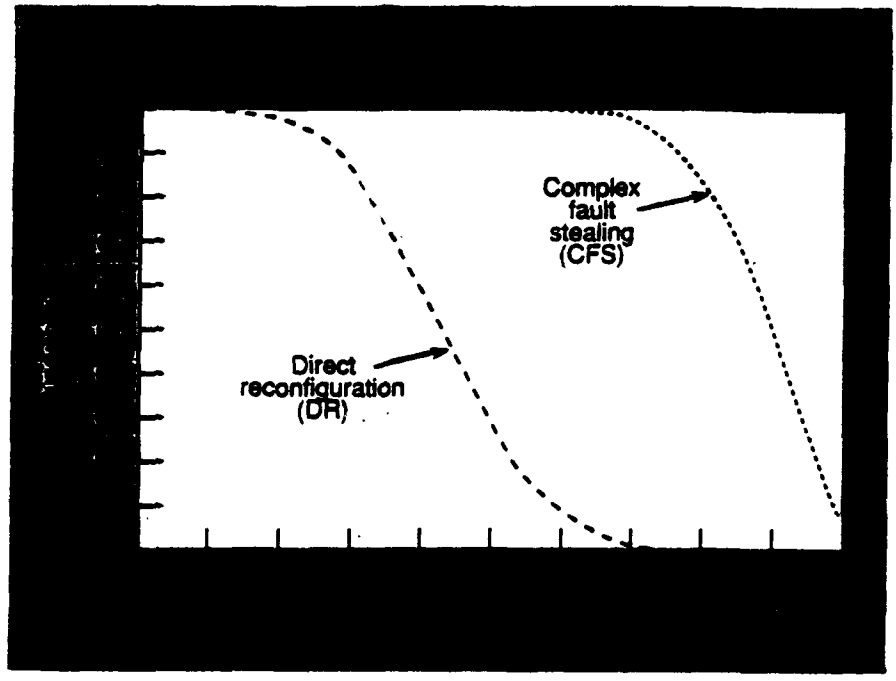


Figure 14. Comparison of array survivabilities (20 × 20).

(1) If $s_i>0$, then the sum of spares in rows 1 through i is greater than the number of faulty cells in rows 1 through i; thus, row i has extra cells available for use by faulty cells in rows $i+1, i+2, ..., M$.

(2) If $s_i<0$, then row i has a deficit and needs to use available cells from rows $i+1, i+2, ..., M$.

(3) If $s_M<0$, then the number of spares is less than the number of faults. In this case, the array is not reconfigurable and the algorithm must exit with failure.

In surplus normalization, the surplus vector is recomputed so that fault shifting can be simplified. For instance, when $s_M<0$, s_M extra cells are available for use by "imaginary" faulty cells in nonexistent rows $M+1, M+2, ...$. Shifting these faulty cells to available cells in row M would be required by the rules used in the fault-shifting phase of FUSS. A better solution is to make $s_M = 0$, which is one of the objectives of surplus normalization.

Fault shifting is the logical replacement of unavailable cells. In FUSS-C, an unavailable cell (i, j) can be (1) shifted down to $(i+1, j)$ if s_i is negative, or (2) shifted up to $(i-1, j)$ if s_{i-1} is positive. After each shift, the corresponding entry in the surplus vector is readjusted toward zero; that is, one is added to s_i in case (1), and one is subtracting from s_{i-1} in case (2). The shifting proceeds until the surplus vector becomes null. Its effect can be described as a cell migration from regions having most faulty cells to regions having less faulty cells.

Cell interconnection is the construction of the logical array. Logical rows and columns are formed one at a time, using a "status matrix" obtained from the fault-shifting phase.

The following example illustrates a FUSS-C reconfiguration. Figure 13a is a matrix A representing a 4 × (4+2) array (4 rows, 4 columns, 2 spare columns) where entry "0" represents a good cell and "1" a faulty cell (column and row indices are added for convenience). This array is equivalent to that used in DR or CFS (see Figure 10) except that the spare row becomes another spare column. The surplus vector obtained after array preprocessing is augmented to the matrix. Figure 13b shows the matrix obtained after fault shifting. The reconfiguration is executed as follows:

(1) Scan the array (Figure 13a) downward. When $s_i<0$, shift a number equal to $|s_i|$ of unavailable cells to row $i+1$ and, when successful, reset s_i to 0. In the example, $s_2 = -1$. Thus, one cell must be shifted down from row 2 to row 3: cell (2,2) is shifted down to cell (3,2), which is now assigned a status code of 3 (Figure 13b); s_2 is reset to 0.

(2) Scan the array upward. When $s_i>0$, shift $|s_i|$ unavailable cells in row $i+1$ to available cells in row i; s_i is reset to 0 when all $|s_i|$ cells are shifted successfully. In the example, $s_3 = 1$. Thus, one cell must be shifted from row 4 to row 3: cell (4,4), the only possible choice, is shifted up to cell (3,4), which assumes the status code of 2; s_3 is readjusted to 0.

At this point, the surplus vector is null, which means that fault shifting is successful. Also, a status matrix B is obtained and shown in Figure 13b. The matrix entries (denoted by $b_{i,j}$) are status codes that guide the cell-interconnection phase of FUSS. Entry $b_{i,j}$ has the following meaning: $b_{i,j} = 0$ if (i, j) is fault free; $b_{i,j} = 1$ if (i, j) is faulty; $b_{i,j} = 2$ if (i, j) replaces $(i+1, j)$; and $b_{i,j} = 3$ if (i, j) is replacing $(i-1, j)$. Since the status of each cell is known, it is easy to automatically derive how cells are interconnected. Afterwards, the corresponding logical array is realized. Figure 13c shows where logical coordinates are given. Figure 11 shows the actual reconfigured array where logical cells along the rows are connected. For FUSS-1, the interconnect requirements are shown in Figure 12, where solid lines are used to connect logical columns and dashed lines are used to connect logical rows. This same switch-bus structure

Table 2. Probability of survival of selected reconfiguration schemes for 20 × 20 arrays (partially from Jervis, Lombardi, and Sciuto, and from Lombardi and Sciuto in "Further reading").

Reconfiguration Scheme	Algorithm Complexity	Survivability When $p = .50$	$p = .75$	$p = 90$
Direct reconfiguration[12]	$O(N)$	0.43	0.35	0.27
Fixed fault stealing[12]	$O(N)$	0.47	0.38	0.30
Complex fault stealing[12]	$O(N)$	0.90	0.84	0.77
Modified index mapping[#@]	$O(N^2)$	0.67	0.62	0.54
Modified optimal reconfiguration algorithm (MORA) (see Lombardi et al.[#@])	$O(N \log N)$	0.90	0.84	0.78
Spanning tree-based[###]	$O(N \log\log N)$	0.93	0.89	0.78
Orthogonal mapping[#@]	$O(N^2)$	0.93	0.88	0.82
Simple full use of suitable spares (FUSS)[7]	$O(N^2)$	1.00	1.00	0.99

#@ See the "Processor-switching techniques" section of "Further reading."

See the "Local-redundancy techniques" section of "Further reading."

can be used for the complex fault-stealing scheme.

In summary, the probability of survival of algorithms in this class improves over the probability of survival of those in the set-exclusion class. Thus, fewer spare cells are being wasted. However, the algorithms become more complex, and the interconnect-link requirements increase. Figure 14 graphs typical values of the probability of survival. Table 2 summarizes survivabilities of all schemes discussed here (or mentioned in "Further reading").

Local-redundancy class

The allocation of redundancy of the reconfiguration schemes in this class is distributed and localized in parts of an array. As mentioned earlier, this is equivalent to partitioning or systematic reduction of an array to smaller subarrays, each of which can be reconfigured independently. A common objective of the schemes in this class is the minimization of the interconnection delays (that is, length of the longest wires between logically adjacent cells). Two typical schemes are discussed here; one illustrates well the partitioning characteristic, and the other is a typical local redundancy scheme.

Divide and conquer. The divide-and-conquer technique[8] consists of procedures for restructuring two-dimensional as well as linear WSI arrays. It focuses on achieving high reconfiguration harvest with short interconnect wires between logically adjacent cells. To construct a two-dimensional array out of functional cells, the array is recursively bisected and the number of live cells in each half is counted. Based on the number of live cells in each half, the algorithm computes the dimensions of the two subarrays that must be constructed and then recursively constructs the subarrays. Each subarray is small enough to be constructed by inspection. The subarrays are then linked to form the complete array. In the worst case, this strategy costs polynomial time and space. The worst-case complexity of the interconnect wires grows with the array size.

Minimization of the maximum wire length and the channel width (number of wires between two rows or two columns) can be achieved by constructing the array out of not all of the fault-free cells but a fraction of them. In other words, the wire length is further reduced if a degraded array harvest is acceptable. This means that some isolated fault-free cells in predominantly defective regions are not used in the reconfiguration.

Interstitial redundancy. The interstitial-redundancy scheme[9] is a good illustration of a local-redundancy technique. The objective of the reconfiguration strategies is to achieve fair area efficiency by using as many good cells as possible, while also ensuring shorter intercell communication links. Here, spare cells are assigned to replace failed primary cells in the prearranged clusters of a two-dimensional array. For instance, in a 25-percent redundancy array, as shown in Figure 15, every spare (squares labeled S) is assigned to a cluster of four primary cells. Each cluster is independent and can tolerate a faulty cell. Since spares are physically close to the cell they replace, restructured intercon-

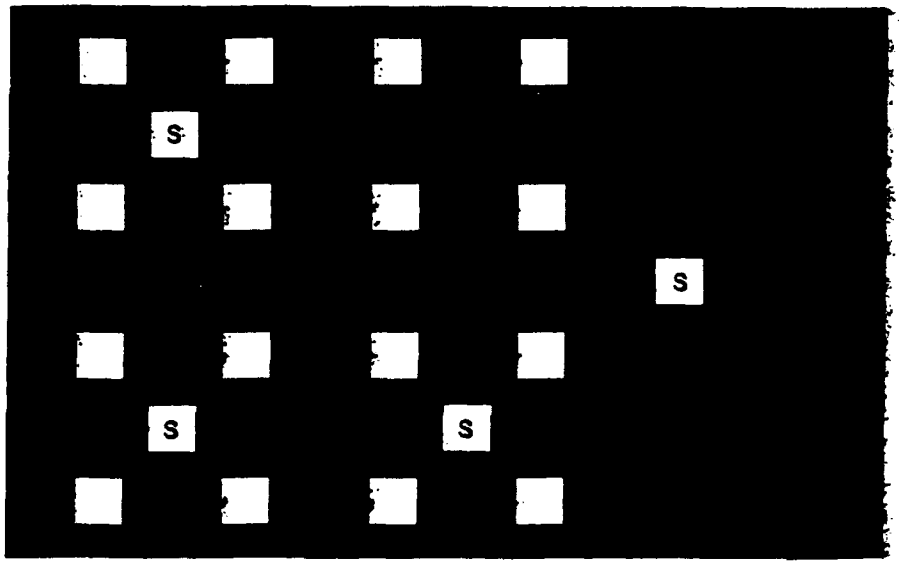


Figure 15. A possible layout for the interstitial redundancy approach to reconfiguration.[9]

nections are fixed and short. This results in low intercell communication delays, thus minimizing performance degradation. For higher redundancy arrays (50 percent, 100 percent, etc.) spares can also be shared among cells in different clusters. For a wide range of array sizes, estimations of yield for arrays with different amounts of redundancy show that approximately 50-percent use of fault-free cells on the chip can be achieved.

Hierarchical redundancy. As mentioned earlier, in local-redundancy schemes, reconfiguration takes place locally in each block of an array partition. If reconfiguration is not possible within a block, the system will fail unless the faulty block can be replaced by a functional one. From a structural point of view, hierarchically combining two or more reconfiguration strategies is possible and practical in some arrays. Typically, these arrays are themselves organized in a hierarchical way. For example, a two-level hierarchical FTPA is an array of small subarrays. A multilevel hierarchical FTPA can be defined recursively, that is, as an array of multilevel subarrays. Different reconfiguration schemes can be applied to different levels of FTPA physical structures, with the overall array reconfiguration schemes having a hierarchical nature. In other words, if a faulty processor cannot be replaced within a subarray (using a given reconfiguration technique) that subarray is declared faulty and possibly replaced by a spare one (possibly using a different reconfiguration technique).

Several examples of hierarchical schemes have been proposed and/or implemented; additional references are given in the local-redundancy techniques section of "Further reading." A typical one is the CHiP (configurable, highly parallel) architecture[10] made up of building blocks, each of which is a two-dimensional CHiP array. The reconfiguration scheme for the high level (array level) is implemented by set switching and that of low level (block level) by processor switching. The CHiP architecture is highly flexible and the switch lattice (connection mechanism) can provide substantial fault tolerance in VLSI/WSI implementation. However, the area complexity of the switch is high.

The analysis and design of hierarchical FTPA structures was considered in Wang and Fortes.[11] It shows that hierarchical architectures can provide much higher reliability than single level, especially in the case of very large arrays.

Time-redundancy approach

The time-redundancy approach to fault-tolerant systems has been used for yield enhancement, reliable computation, and self testing. Ideally, a time-redundancy reconfiguration scheme uses only logical spare cells, that is, physical cells that perform their own functions as well as the functions of the faulty ones. However, it can also include techniques that use some (limited) physical spare cells in the reconfiguration. Time-redundancy techniques can also be used to support graceful degradation, because they can prevent system failure after spares are exhausted while reducing the computational speed of the overall system. A time-redundancy scheme can use a limited number of cells to perform operations that would require many more cells in a hardware-redundancy scheme.

Sami and Stefanelli[12] presented a typical time-redundancy technique that involves repeated use of functional cells and requires two phases and three clock cycles. Each cell is associated with two logical cells. During the first phase, the cell executes the functions of the first logical cell; during the second phase, it executes the functions of the second logical cell. The redundant hardware elements required by the techniques are much less than those required by hardware approaches discussed previously. However, the processing speed is lowered.

SRE and ARCE. The successive row elimination (SRE) and alternate row column elimination (ARCE) schemes[13] are examples of graceful degradation schemes. In SRE, a row is eliminated if it contains one or more faults (set switching), and in ARCE either a row or a column that contains one or more faulty elements is bypassed. These schemes are similar to the Kuo-Fuchs scheme presented earlier, except that no spare rows or spare columns are used and they are complemented by algorithm reconfiguration techniques that allow an algorithm designed for a processor array of a given size to run on a processor array of smaller size. Rows and columns can be theoretically eliminated until one row and one column are left in the array. The performance of the algorithms is allowed to degrade gracefully; for example, with up to $(N/2)-1$ faults, SRE allows the array to operate at half the speed of the fully operational array.

In summary, time-redundancy approaches to array reconfiguration might require less hardware but demand longer processing time than hardware-redundancy techniques. These techniques can be particularly useful in some architectures, such as systolic arrays, where idle cycles can be used by logical spares to perform functions of the faulty cells. In addition, the time-redundancy approach can be considered when decreasing speed is an acceptable form of degradation.

Concluding remarks

The proposed taxonomy for classifying different reconfiguration schemes for processor arrays can be used to compare and contrast many possible approaches in light of basic orthogonal characteristics. In addition, we hope the different schemes presented to illustrate the taxonomy will help interested engineers and researchers understand existing approaches and develop new ones.

An important yet difficult question to be faced by designers is that of deciding which of the many possible schemes is the best for their application. The ultimate benefit of reconfiguration schemes is yield improvement (in the case of static reconfiguration) or reliability increase (for dynamic reconfiguration).

A designer must be able to evaluate the impact of a scheme on yield and/or reliability, and these are complex functions of technology, application, scheme survivability, and scheme overhead. The latter factor includes the hardware, software, and time necessary to implement the scheme. It can be the cause for faults and performance degradation that mitigate the survivability of the scheme. Reliability and yield models must account not only for processing elements but also for connections, switches, reconfiguration logic, and other hardware and software mechanisms.

Much research is currently under way to identify valid defect and fault-distribution models and to develop models and tools for reliability and yield evaluation. Studies by the authors and other researchers clearly show that there is no universally ideal reconfiguration scheme. For this reason, identifying analytical approaches to the evaluation of survivability, reliability, and yield has become an important research direction. Good results in this area will support a deeper technical understanding of reconfiguration schemes and systematic derivation, analysis, and evaluation of new

Table 3. Comparison of reconfiguration classes in terms of SEAL (simplicity, efficiency, area, locality) figure of merit.

Scheme	Simplicity	Efficiency	Area	Locality
Set switching	Good	Poor	Poor	Good
Processor switching	Fair	Good	Fair	Good
Local redundancy	Good	Fair	Fair	Good
Time redundancy	Poor	Good	Good	Good

approaches. This will help explain empirical approaches and replace ad hoc solutions by fundamental techniques.

As mentioned above, it is difficult to make universal statements about the relative advantages of the different schemes, namely the set-switching class, processor-switching class, local-redundancy class, and time-redundancy class. However, in Table 3, we attempt to compare, in general terms, the five major classes in relation to the SEAL (simplicity, efficiency, area, locality) figures of merit. Some general issues facing the schemes are also briefly reviewed below.

From a redundancy point of view, the local-redundancy class of reconfiguration schemes aims at achieving short wires, thus short communication delays, for interconnecting logically adjacent cells in the reconstructed arrays. This is accomplished by either physically assigning spare cells to a group of cells or by systematically partitioning the array. Minimizing the wire lengths necessitates some waste of spare cells, which can result in suboptimal reconfiguration harvest or underutilization of spares.

The waste of redundant resources can be substantial in the set-switching class of reconfiguration schemes because entire sets of cells are replaced. In set switching, interconnection and control circuitry are much simpler than those in other classes. For this reason, schemes could be easily implemented. On the other hand, the waste of spare cells makes the techniques in this class impractical in some applications.

The processor-switching class of reconfiguration techniques aims at minimizing the longest wire length connecting two logically adjacent cells and, at the same time, minimizing the waste of spares. This requires more complex algorithms and more involved interconnect and control reconfiguration circuitry. Existing schemes demonstrate better spare use (than those in the two previously discussed classes) with polynomial reconfiguration time complexity. However designers must carefully explore the design of interconnection and control hardware before efficient implementation can be realized.

Finally, when performance degradation is acceptable, physical redundancy can be replaced by time redundancy to improve the silicon area in FTPAs. Reconfiguration schemes that employ this strategy are grouped in the time-redundancy class. Here, cells execute not only their tasks, but also the tasks of their faulty neighbors. For this reason, these schemes can result in degraded computational speed.

We have overviewed, characterized, and classified some typical reconfiguration schemes in light of a proposed taxonomy. This taxonomy can be used as a guide for future research in design and analysis of reconfiguration schemes.

Studying how to evaluate fault-tolerant arrays and how to exploit application characteristics to achieve dependable computing are important complementary directions of research towards reliable processor-array design.

A related research problem is that of functional reconfiguration, that is, learning how to configure the topology of a parallel system to implement a different function or run a different application. This topic transcends the scope of this article, but it is worthwhile stating that results in this area are relevant and useful to the problem of reconfiguration for fault tolerance and vice versa.

In fact, important directions of research include how to apply or extend processor-array reconfiguration algorithms to other topologies and how to marry functional and fault-tolerance reconfiguration requirements and solutions. The Diogenes approach discussed in this article is an interesting case where this goal is naturally achieved. ■

Acknowledgements

We thank the reviewers and *Computer* Editor-in-chief Bruce Shriver for their comments and suggestions on how to improve the organization and content of this article. In addition, discussions with Noe Lopez-Benitez and Yi-Xiang Wang helped enhance the article. We also thank Clara C. Alves for her help in preparing the manuscript.

This work was partially supported by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contracts No.00014-85-k-0588 and No.00014-88-k-0723.

References

1. J.A. Abraham et al., "Fault Tolerance Techniques for Systolic Arrays," *Computer*, Vol. 20, No. 7, July 1987, pp. 65-75.
2. H.T. Kung and M. Lam, "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays," *J. Parallel and Distributed Computing*, Vol. 1, Issue 1, 1984, pp. 32-63.
3. I. Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," *Proc. 8th Symp. Computer Architecture*, CS Press, Los Alamitos, Calif., Order No. 346 (microfiche), 1981, pp. 425-442.
4. S.-Y. Kuo and W.K. Fuchs, "Efficient Spare Allocation for Reconfigurable Arrays," *IEEE Design & Test*, Feb. 1987, pp. 24-31.
5. A.L. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors," *IEEE Trans. Computers*, Vol. C-32, Oct. 1983, pp. 902-910.
6. R. Negrini, M. Sami, and R. Stefanelli, "Fault-Tolerance Techniques for Array Structures Used in Supercomputing," *Computer*, Vol. 19, No. 2, Feb. 1986, pp. 78-87.
7. M. Chean and J.A.B. Fortes, "FUSS: A Reconfiguration Scheme for Fault-Tolerant Processor Arrays," *Int'l Workshop Hardware Fault Tolerance in Multiprocessors*, June 1989, pp. 30-32.
8. T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. Computers*, Vol. C-34, May 1985, pp. 448-461.
9. A.D. Singh, "Interstitial Redundancy: An Area Efficient Fault-Tolerance Scheme for Large Area VLSI Processor Arrays," *IEEE Trans. Computers*, Vol. C-37, No. 11, Nov. 1988, pp. 1,398-1,410.
10. K.S. Hedlund and L. Snyder, "Wafer-Scale Integration of Configurable, Highly Parallel (CHiP) Architectures and Processors"

(extended abstract), *Proc. 1982 Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 421 (microfiche), 1982, pp. 262-264.

11. Y.-X. Wang and J.A.B. Fortes, "On the Analysis and Design of Hierarchical Fault-Tolerant Processor Arrays," *Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, Oct. 1988, pp. 347-355.

12. M.G. Sami and R. Stefanelli, "Fault Tolerance of VLSI Processing Arrays: The Time-Redundancy Approach," *Real-Time Systems Symp.*, Dec. 1984, pp. 200-207.

13. J.A.B. Fortes and C.S. Raghavendra, "Gracefully Degradable Processor Arrays," *IEEE Trans. Computers*, Vol. C-34, Nov. 1985, pp. 1,033-1,044.

Further reading

Set-switching techniques

A set-switching reconfiguration scheme replaces a faulty cell by logically removing a set of cells (row, column, block, etc.) that contains the faulty cell. For this reason, the scheme can be easily implemented; however, the waste of functional cells is large since many of them cannot be used. For instance, if an array has R spare rows and every time a fault occurs the row containing the fault is replaced by one spare row, as few as $R+1$ faulty cells on different rows can make the array useless. For additional reading, see discussion in the main text and schemes proposed in the following references:

Batcher, K.E., "Design of a Massively Parallel Processor," *IEEE Trans. Computers*, Vol. C-29, Sept. 1980, pp. 836-840.

McCanny, J.V., and J.G. McWhirter, "Yield Enhancement of Bit-Level Systolic Array Chips Using Fault-Tolerant Techniques," *Electronics Letters*, Vol. 19, July 1983, pp. 525-527.

Ishikawa, T., et al., "Hierarchical Array Processor (HAP) Featuring High Reliability and High System Performance," *1986 Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 724 (microfiche), Aug. 1986, pp. 293-300.

Kim, J.H., and S.M. Reddy, "On Easily Testable and Reconfigurable Two-Dimensional Systolic Arrays," *Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 783, Aug. 1987, pp. 101-109.

Hedlund, K.S., "WASP — A Wafer-Scale Systolic Processor," *IEEE Int'l Conf. Computer Design*, CS Press, Los Alamitos, Calif., Order No. 642 (microfiche), Oct. 1985, pp. 665-671.

Processor-switching techniques

The replacement strategy used in these schemes proceeds in a chain fashion such that a faulty cell is replaced by (shifted to) an immediate neighbor, this neighbor is replaced by its neighbor, and so on until the spare cell is reached. The sequence of replacements is often called a replacement chain or a compensation path (see Kung, Jean, and Chang in the "Local redundancy techniques" section below). In a more complex scheme, a faulty cell can also be shifted to a distant neighbor. Many schemes have been proposed. Some are discussed in the main text, and others are described in Kuo and Fuchs[4] and the following references:

Ciminiera, L., C. Demartini, and A. Valenzano, "Defect-Tolerant Array Structures for VLSI and WSI Architectures," *Proc. 20th Hawaii Int'l Conf. System Science*, 1987, pp. 21-30.

Jervis, L., F. Lombardi, and D. Sciuto, "Orthogonal Mapping: A Reconfiguration Strategy for Fault-Tolerant VLSI/WSI 2-D Arrays," *Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, Oct. 1988, pp. 309-318.

Lombardi, F., R. Negrini, and R. Stefanelli, "Reconfiguration of VLSI Arrays: A Covering Approach," *17th Int'l Symp. Fault-Tolerant Computing Systems*, CS Press, Los Alamitos, Calif., Order No. 778 (microfiche), 1987, pp. 251-256.

Distante, F., M.G. Sami, and R. Stefanelli, "A General Index Mapping Technique for Array Reconfiguration," *Int'l Symp. Circuits and Systems*, June 1988, pp. 559-563.

Youn, H.Y., and A.D. Singh, "A Highly Efficient Design for Reconfiguring the Processor Array in VLSI," *Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 889 (microfiche), Aug. 1988, pp. 375-382.

Local-redundancy techniques

In these schemes, a given spare processor can be used to replace only one of a predetermined subset of cells of the array, typically in physical proximity to the spare. Many of these schemes are implemented in a hierarchical array structure (see Ishikawa in the "Set-switching techniques" section; and see Jesshope and Bentley, Reeves, and Hwang and Raghavendra below). The "partitioning scheme" (see Kung, Jean, and Chang below) and the "spanning tree" scheme are similar to Leighton and Leiserson's divide-and-conquer strategy.[8] The spanning tree approach chooses to partition an FTPA along the array diagonal, and the subarray construction is implemented following a spanning tree algorithm. Other schemes are somewhat similar to the CHiP approach discussed in the text. References for these schemes are as follows:

Jesshope, C., and L. Bentley, "Techniques for Implementing Two-Dimensional Wafer-Scale Processor Arrays," *IEE Proc.*, Vol. 134, Pt. E, Mar. 1987, pp. 87-92.

Reeves, A.P., "Fault Tolerance in Highly Parallel Mesh-Connected Processors," in *Computing Structure for Image Processing*, M.J.B. Duff, ed., Academic Press, London, 1983, pp. 77-94.

Hwang, J.H., and C.S. Raghavendra, "VLSI Implementation of Fault-Tolerant Systolic Arrays," *IEEE Int'l Conf. Computer Design*, CS Press, Los Alamitos, Calif., Order No. 735, 1986, pp. 110-113.

Kung, S.Y., S.N. Jean, and C.W. Chang, "Fault-Tolerant Array Processors Using Single-Track Switches," *IEEE Trans. Computers*, Vol. 38, No. 4, Apr. 1989, pp. 501-514.

Lombardi, F., and D. Sciuto, "Reconfiguration in WSI Arrays Using Minimum Spanning Trees," *CompEuro 87*, CS Press, Los Alamitos, Calif., Order No. 773, May 1987, pp. 547-550.

Time-redundancy approaches

In these schemes, the redundant hardware incorporated into the FTPA is limited to circuitry that enables functional cells to execute repeated operations. Time-redundant schemes other than those discussed in the text can be found

·n the first three references listed below. Time-redundancy approaches have also been used extensively for error detection, as discussed in the final two references listed below.

Negrini, R., and R. Stefanelli, "Comparative Evaluation of Space- and Time-Redundancy Approaches for WSI Processing Arrays," in *Wafer-Scale Integration*, G. Saucier and J. Trilhe, eds., Elsevier Science Publishers B.V. (North Holland), 1986, pp. 207-222.

Fortes, J.A.B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays," in *Systolic Arrays*, W. Moore, A. McCabe, and R. Urquhart, eds., Adam Hilger, 1986, pp. 259-268.

Siewiorek, D.P., and R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.

Johnson, B.W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, Reading, Mass., 1989.

Negrini, R., and R. Stefanelli, "Time Redundancy in WSI Arrays of Processing Elements," *Proc. Int'l Conf. Supercomputing Systems*, CS Press, Los Alamitos, Calif., Order No. 654 (microfiche), 1985, pp. 429-438.

Mengly Chean is an associate research computer scientist at the Bellaire Research Center, Shell Development Company, Houston, Texas. He served as a graduate instructor and research assistant at Purdue University, West Lafayette, Indiana, from 1984 to 1989. His research interests include computer architecture, fault-tolerant computing, parallel processing, and computer networks.

Chean received his BSEE in 1979 from the University of Maryland. He received his MSEE in 1980 and his PhD in computer engineering in 1989, both from Purdue. He is a member of the IEEE, the IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

Fortes can be contacted at the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

Jose A.B. Fortes has been on the faculty of Purdue University since 1984 and is currently an associate professor at the Purdue School of Electrical Engineering. He is on a one-year leave through July 1990 at the National Science Foundation, where is serving as a program director for microelectronic systems architecture. His technical interests are in the areas of parallel processing and fault-tolerant computing, and he has had more than 50 papers published.

Fortes received MS and PhD degrees in electrical engineering from Colorado State University and the University of Southern California, respectively. He is on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *Journal of VLSI Signal Processing*, is co-chair of the Program Committee for the 1990 International Conference on Application Specific Array Processors (to be held in September), and is a member of the IEEE and the IEEE Computer Society.

REFERENCE NO. 3

Shang, W., and Fortes, J. A. B., "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, Volume 40, Number 6, June 1991, pp. 723-743.

Note - This paper extends the results of reference [1] to the case when *several* outputs must be computed in the minimal amount of time.

Time Optimal Linear Schedules for Algorithms with Uniform Dependencies

Weijia Shang, *Student Member, IEEE*, and Jose A. B. Fortes, *Member, IEEE*

Abstract—**An algorithm can be thought of as a set of indexed computations and if one computation uses data generated by another computation then this data dependence can be represented by the difference of their indexes (called *dependence vector*). Many important algorithms are characterized by the fact that data dependencies are *uniform*, i.e., the values of the dependence vectors are independent of the indexes of computations. *Linear schedules* are a special class of schedules described by a linear mapping of computation indexes into time. This paper addresses the problem of identifying optimal linear schedules for uniform dependence algorithms so that their execution time is minimized. Procedures are proposed to solve this problem based on the mathematical solution of a nonlinear optimization problem. The complexity of these procedures is independent of the size of the algorithm. Actually, the complexity is exponential in the dimension of the index set of the algorithm and, for all practical purposes, very small due to the limited dimension of the index set of algorithms of practical interest. The results reported in this paper can be used to derive time-optimal systolic designs and applied in optimizing compilers to restructure programs at compile-time in order to maximally exploit available parallelism.**

Index Terms— **Algorithm mapping, data dependency, linear schedule, optimizing compiler, nested-loop program, systolic array, time-optimal.**

I. Introduction

A DIFFICULT problem in the parallel execution of an algorithm is the choice of a schedule which results in minimum execution time. This paper formulates and solves this problem for a particular class of algorithms and a specific type of schedules denoted *linear schedules*. The algorithms under consideration are characterized by *uniform data dependencies* and *unit-time computations;* they include those described by single uniform recurrences [1] and resemble a very large number of systolic computations and algorithms described by programs with nested loops. The results reported in this paper can be used to derive time-optimal systolic designs for those algorithms and can also be applied in optimizing compilers [13], [19] to restructure programs at compile-time in order to maximally exploit available parallelism.

Manuscript received March 15, 1989; revised January 10, 1990. This work was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contracts 00014-85-k-0588 and 00014-88-k-0723.

W. Shang is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9100073.

In simple terms, an algorithm is represented in this paper as a partially ordered subset of a multidimensional integer lattice (called *index set*). The points of this lattice correspond to (or index) computations and the partial order reflects the data dependencies between them. These data dependencies are represented as vectors that connect points of the lattice. Informally, if a given dependence vector is always present when the vector difference between any two lattice points equals the dependence vector, then the dependence is said to be *uniform*. If all dependences are uniform then the algorithm is said to be a *uniform dependence algorithm*. *A linear schedule* is a mapping from the multidimensional algorithm index set into the one-dimensional time space; this mapping is expressed as a linear transformation that involves the multiplication of a vector, called *linear schedule vector,* by each and every point of the index set. The image of the index point under the mapping is the time of execution of the computation indexed by that point. The purpose of this paper is to show how the linear schedule vector can be determined so that the algorithm can be executed in a minimal amount of time (achievable by a linear schedule). This algorithm model and the notion of linear schedule are easily related to similar models and concepts used in [1]–[13], [19], and several other works.

A *free schedule* schedules computations to execute as soon as their operands are available. The total execution time that results from using a free schedule is the exact lower bound for the execution time of the algorithm. The difference between the execution time achieved by the free schedule and the execution time achievable by an optimal linear schedule is bounded by a constant [1]. In [16], a class of algorithms is identified for which that difference is zero, i.e., linear schedules can be as fast as the free schedule and achieve the execution time lower bound for this class of algorithms.

In practice, linear schedules are often characterized by the fact that computations in one or more "computational wavefronts" take place at any given instant of time. These types of schedules have been proposed for the execution of many practical algorithms including the solution of differential equations [20], the computation of uniform recurrences [1], matrix arithmetic algorithms [8], [21], [22], digital signal processing applications [8], [22], [23], algorithms described by nested-loop programs [19], [21], and systolic array algorithms [8], [21]. Linear schedules can be described conveniently by using special purpose or extended programming languages [8], [19], [21], formal mathematical descriptions [1]–[12], or English-like explanations [23]. In some of these references, optimization procedures were proposed for the determination

0018-9340/91/0600-0723$01.00 © 1991 IEEE

of optimal linear schedules; however, they either have a heuristic nature or have a more restricted applicability than the method described in this paper. In Section IV, one of these procedures, referred to as *linear programming approach*, is presented and compared to that proposed in Section III. The linear programming approach can be directly derived from Lemmas 1–4 in this paper and is similar or can be related to the approaches proposed in [19], [27], [28], and other references.

This paper is organized as follows. The basic terminology and definitions used throughout the paper are introduced in Section II as well as the formulation of the problem of determining an optimal linear schedule. Section III is dedicated to the description of the optimization procedure that finds the time-optimal linear schedule for any given algorithm with uniform dependencies, the proof of its correctness and the analysis of its complexity. Section IV presents the linear programming approach and compares it to the approach presented in Section III. A particular class of algorithms for which the proposed solution is greatly simplified is considered in Section V and the corresponding simpler optimization procedure is also provided. Section VI is dedicated to conclusions and future work.

II. Terminology and Definitions

Throughout this paper, *sets*, *matrices*, and *row vectors* are denoted by capital letters, *column vectors* are represented by lower case symbols with an overbar, and *scalars* correspond to lower case letters. The *transposes* of a vector $\bar{v}$ and a matrix M are denoted $\bar{v}^T$ and M^T, respectively. The symbol E_i denotes the row vector whose entries are all zeros except that the ith entry is equal to unity. The vector $\bar{1}$ (or $\bar{0}$) denotes the row vector or column vector whose entries are all ones (or zeroes). The dimensions of vectors $\bar{1}$ and $\bar{0}$ and whether they denote row or column vectors are implied by the context in which they are used. The vector space spanned by a set of vectors $S = \{\bar{v}_1, \bar{v}_2, \cdots, \bar{v}_k\}$ is denoted $sp\{\bar{v}_1, \bar{v}_2, \cdots, \bar{v}_k\} = sp\{S\}$ and its dimension (i.e., the number of linearly independent vectors in S) is denoted $\dim\{S\}$. The symbol I denotes the identity matrix. The rank of a matrix A is denoted $\mathrm{rank}(A)$. The set of rational numbers, the real space, and the set of integers are denoted Q, $\mathbf{R}$, and Z, respectively. The set of nonnegative integers and the set of positive integers are denoted N and N^+, respectively. The empty set is denoted $\varnothing$. The notation $|S|$ represents the cardinality of set S. Let $\bar{v}$ and $\bar{u}$ be two vectors. Then $\bar{v} \geq \bar{u}$ means every component of $\bar{v}$ is greater than or equal to the corresponding component of $\bar{u}$. As a final remark, if x is an element of a set S, the notation $x \in S$ is used and this notation is also used to indicate that a column vector $\bar{m}_j$ (or row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j \in M$ ($M_i \in M$) means $\bar{m}_j (M_i)$ is a column (row) vector of matrix M.

The algorithms of interest in this paper are the so-called uniform dependence algorithms defined as follows.

Definition 2.1 (Uniform dependence algorithm): A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\bar{j}) = g_{\bar{j}}(v(\bar{j} - \bar{d}_1), v(\bar{j} - \bar{d}_2), \cdots, v(\bar{j} - \bar{d}_m)) \quad (2.1)$$

where

1) $\bar{j} \in J \subset Z^n$ is an index point (a column vector), J is the *index set* of the algorithm, and $n \in N^+$ is the number of components of $\bar{j}$;
2) $g_{\bar{j}}$ is the computation indexed by $\bar{j}$, i.e., a single-valued function computed "at point $\bar{j}$" in a *single unit of time;*
3) $v(\bar{j})$ is the value computed "at $\bar{j}$", i.e., the result of computing the right-hand side of (2.1) and
4) $\bar{d}_i \in Z^n$, $i = 1, \cdots, m$, $m \in N$ are *dependence vectors*, also called *dependencies*, which are constant (i.e., independent of $\bar{j} \in J$); the matrix $D = [\bar{d}_1, \cdots, \bar{d}_m]$ is called the *dependence matrix*.

The class of uniform dependence algorithms is a simple extension of the class of computations described by uniform recurrence equations [1]. The main difference is that uniform dependence algorithms allow for different functions to be computed (in a unit of time) at different points of the index set. From a practical viewpoint, uniform dependence algorithms can be easily related to programs where 1) a single statement appears in the body of a multiply-nested loop and 2) the indexes of the variable in the left-hand side of the statement differ by a constant from the corresponding indexes in each reference to the same variable in the right-hand side. Alternative computations can occur in each iteration as a result of a single conditional statement as long as data dependencies do not change. Nested loop programs with multiple statements can also use the techniques of this paper together with the alignment method discussed in [24] and [25].

For the purpose of this paper, only structural information of the algorithm, i.e., the index set J and the dependence matrix D, is needed. Other information, such as what computations occur at different points and where and when input/output of variables takes place, can be ignored. Therefore, a uniform dependence algorithm with index set J and dependence matrix D is hereon characterized simply by the pair (J, D). Herein, it is assumed that, as in Definition 2.1, the letters n and m always denote the dimension of index points in J and the number of dependence vectors, respectively. While no restrictions apply to the type of the index set of an algorithm, the following is assumed in regard to how index sets are described.

Assumption 2.1 (Index set, index constraint matrix, size vector, and convex hull of the index set): The index set $J \subset Z^n$ of any given algorithm is described as a set of integer points (vectors) whose convex hull R [14, p. 35] is a nondegenerate (explained in the following paragraph) polyhedron, i.e.,

$$R = \{\bar{x} : A\bar{x} \leq \bar{b}, A \in Z^{a \times n}, \bar{b} \in Z^a, \bar{x} \in \mathbf{R}^n, a \in N^+\} \quad (2.2)$$

and

$$J \subseteq \{\bar{j} : \bar{j} \in R \wedge \bar{j} \in Z^n\}.$$

Matrix A is called *index constraint matrix of* J; vector $\bar{b}$ is called *size vector;* R is called *convex hull* of index set J.

Equation (2.2) simply states that R is a polyhedron. If R contains $n+1$ points $\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_{n+1}$ such that $\bar{x}_1 - \bar{x}_{n+1}$, $\bar{x}_2 - \bar{x}_{n+1}, \cdots, \bar{x}_n - \bar{x}_{n+1}$ are linearly independent, then R is *nondegenerate*. For example, a line and a square with nonzero area are cases of degenerate and nondegenerate polyhedra in $\mathbf{R}^2$, respectively. Clearly, degenerate polyhedra in $\mathbf{R}^n$ can be reexpressed as nondegenerate polyhedra in $\mathbf{R}^{n'}$ for some $n' < n$. According to Assumption 2.1, R is not necessarily bounded and J is not necessarily *dense* (J is not dense if there is at least one integer point in R that does not belong to J). Extreme points of polyhedron R are always integers and belong to the index set J because R is the convex hull of index set J. Finally, different size vectors correspond to instances of the same algorithm that differ only in their sizes (but have the same shape). The following example illustrates the concepts introduced in Definition 2.1 and Assumption 2.1.

Example 2.1: Consider the following uniform dependence algorithm:

$$v(j_1, j_2) = g(v(j_1 - 2, j_2 - 2), v(j_1, j_2 - 3)) \quad \text{where}$$
$$j_1 = 0, \cdots, 2s$$
$$j_2 = \begin{cases} 0, \cdots, 2s - j_1 & \text{if } s \le j_1 \le 2s \\ s - j_1, \cdots, 2s - j_1 & \text{if } 0 \le j_1 \le s \end{cases}, s \in N^+.$$

The index set J of this algorithm is shown in Fig. 1. The reader can verify that it can be described as $J = \{\bar{j} = [j_1, j_2]^T : A\bar{j} \le \bar{b}, \bar{j} \in Z^2\}$ and its convex hull is $R = \{\bar{x} = [x_1, x_2]^T : A\bar{x} \le \bar{b}, \bar{x} \in \mathbf{R}^2\}$, where

$$A = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix}, \qquad \bar{b} = \begin{bmatrix} 0 \\ 0 \\ 2s \\ -s \end{bmatrix}.$$

There are four extreme points, i.e., $\bar{e}_1 = [0, 2s]^T$, $\bar{e}_2 = [0, s]^T$, $\bar{e}_3 = [s, 0]^T$, and $\bar{e}_4 = [2s, 0]^T$. The index set J in this example is bounded, dense, and nondegenerate. The dependence matrix is $D = [\bar{d}_1, \bar{d}_2]$ where $\bar{d}_1 = [2, 2]^T$ and $\bar{d}_2 = [0, 3]^T$. End of example.

In the example above, the index set J is bounded by four lines described by equations $x_1 = 0$, $x_2 = 0$, $x_1 + x_2 = s$, and $x_1 + x_2 = 2s$, respectively. Each line is called a boundary surface of index set J and the definitions of this and other ancillary concepts are formally stated next.

Definition 2.2 (Boundary surface, norm vector, norm space, and norm intersection): Let $A \in Z^{a \times n}$ be the index constraint matrix of algorithm (J, D) and let $A_{h_1}, \cdots, A_{h_k} \in A$, $k \le \min\{a, n\}$, be linearly independent rows of A. The hyperplane $F = \{\bar{x} : A_{h_i}\bar{x} = b_{h_i}, \bar{x} \in \mathbf{R}^n, i = 1, \cdots, k\}$ is called an *$(n-k)$-dimensional boundary surface* of J if $F \cap R \neq \emptyset$. The row vectors $A_{h_1}, \cdots, A_{h_k}$ are called the *norm vectors* of boundary surface F, the notation $\text{surf}\{A_{h_1}, \cdots, A_{h_k}\}$ is used to represent F and $sp\{A_{h_1}, \cdots, A_{h_k}\}$ is called the *norm space* (of index set J) associated with F. The intersection of any two norm spaces of J is called a *norm intersection* of index set J.

When $k = n$ in the last definition, the corresponding surface is "zero-dimensional," i.e., it is an extreme point of J. The norm space spanned by norm vectors $A_{h_1}, \cdots, A_{h_k}$ is "*perpendicular*" to its boundary surface F, i.e., if V is a row vector in $sp\{A_{h_1}, \cdots, A_{h_k}\}$, then for all $\bar{x}_1, \bar{x}_2$ in $F = \{\bar{x} : A_{h_i}\bar{x} = b_{h_i}, \bar{x} \in \mathbf{R}^n, i = 1, \cdots, k\}$, $V(\bar{x}_1 - \bar{x}_2) = 0$. For a given index set J with index constraint matrix $A \in Z^{a \times n}$, there are at most $\sum_{k=1}^{\min\{a,n\}} \binom{a}{k}$ distinct norm spaces.

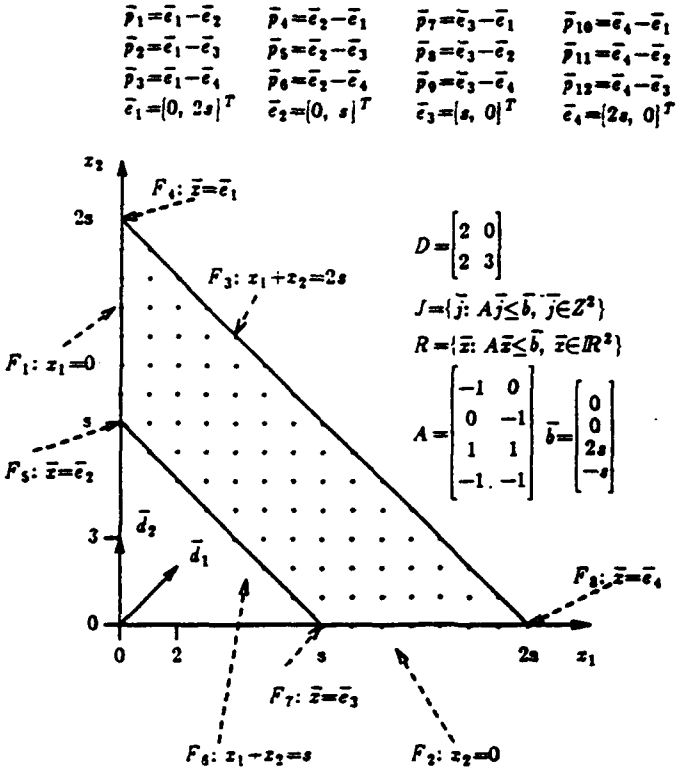


Fig. 1. The index set of the algorithm of Example 2.1. Each point corresponds to a computation.

Example 2.2: In Example 2.1, $A_1 = [-1, 0]$, $A_2 = [0, -1]$, $A_3 = [1, 1]$, $A_4 = [-1, -1]$, $b_1 = 0$, $b_2 = 0$ and $b_3 = 2s$. As shown in Fig. 1, $F_1 = \{\bar{x} : A_1\bar{x} = b_1, \bar{x} \in \mathbf{R}^2\} = \{\bar{x} : x_1 = 0, \bar{x} \in \mathbf{R}^2\}$ is the one-dimensional boundary surface of J containing the points on the x_2 axis. Also, $F_4 = \{\bar{x} : A_1\bar{x} = b_1, A_3\bar{x} = b_3, \bar{x} \in \mathbf{R}^2\} = \{\bar{x} : x_1 = 0, x_1 + x_2 = 2s, \bar{x} \in \mathbf{R}^2\}$ is a "zero"-dimensional surface of J that corresponds to the extreme point $\bar{e}_1 = [0, 2s]^T$. Notice that the set $F = \{\bar{x} : A_1\bar{x} = b_1, A_2\bar{x} = b_2, \bar{x} \in \mathbf{R}^2\} = \{\bar{x} : x_1 = x_2 = 0\}$ is not a boundary surface because $F \cap R = \emptyset$. $F' = \{\bar{x} : A_3\bar{x} = b_3, A_4\bar{x} = b_4, \bar{x} \in \mathbf{R}^2\}$ is not a boundary surface because A_3 and A_4 are linearly dependent. One norm intersection of J is the intersection $sp\{A_1\} \cap sp\{A_1, A_3\} = sp\{A_1\} = \{X = [x_1, x_2] : x_2 = 0, X \in \mathbf{R}^{1\times 2}\}$ where A_1, A_3, and X are row vectors, i.e., the intersection of the norm space of F_1 and the norm space of F_4. End of example.

Definition 2.3 (Schedules): A *schedule* for algorithm (J, D) is a function $\sigma : J \to Z$ such that for any arbitrary index points $\bar{j}, \bar{j}' \in J$, $\sigma(\bar{j}) < \sigma(\bar{j}')$ if $\bar{j}' = \bar{j} + \bar{d}_i$, $\bar{d}_i \in D$.

In other words, a schedule is a mapping which assigns a time of execution to each computation of the algorithm in such a way that dependencies are preserved, i.e., if the computation indexed by $\bar{j}'$ depends on the computation indexed by $\bar{j}$, then the computation indexed by $\bar{j}'$ can be executed only after the execution of the computation indexed by $\bar{j}$. The schedules of

interest in this paper are so-called linear schedules which are defined next.

Definition 2.4: For algorithm (J, D) a schedule $\sigma_\Pi: J \to Z$ is a *linear schedule* if $\sigma_\Pi(\vec{j}) = \lfloor \Pi\vec{j} + c \rfloor$,[1] $\vec{j} \in J$, where $\Pi \in Q^{1\times n}$ is such that $\min\{\Pi\vec{d}_i : \vec{d}_i \in D\} = 1$ and $c = -\min\{\Pi\vec{j} : \vec{j} \in J\}$.

Definitions 2.3 and 2.4 are similar to equivalent concepts in [1]. In the last definition, the entries of Π can be any rational number. However, given such a Π, it is always possible to find $\Pi' \in Z^{1\times n}$ and a constant $disp\Pi' \in N^+$ such that $\Pi = \Pi'/disp\Pi'$ where $disp\Pi'$ equals the least common multiple of the denominators of the entries of Π. For this reason, the following definition of a linear schedule is equivalent to that of Definition 2.4 and is used throughout this paper.

Definition 2.5 (Linear schedule and linear schedule vector): For algorithm (J, D) a *linear schedule* is a function $\sigma_\Pi: J \to N$ such that

$$\sigma_\Pi(\vec{j}) = \lfloor (\Pi\vec{j} + c)/disp\Pi \rfloor, \qquad \vec{j} \in J \tag{2.3}$$

where $\Pi \in Z^{1\times n}$, $disp\Pi = \min\{\Pi\vec{d}_i : \vec{d}_i \in D\} > 0$, $gcd(\pi_1, \cdots, \pi_n)^2 = 1$ and $c = -\min\{\Pi\vec{j} : \vec{j} \in J\}$. The row vector Π is called *linear schedule vector* associated with σ_Π.

Example 2.3: Fig. 2 depicts a linear schedule $\sigma_\Pi = \lfloor ([1,1]\vec{j} - 5)/3 \rfloor$ for the algorithm of Example 2.1 ($s = 5$) where $\Pi = [1, 1]$, $c = -5$ and $disp\Pi = 3$. The hyperplanes (dotted lines in this case) described by the equations $\Pi\vec{j} = c'$ for different values of c' help the visualization of the schedule because all points contained in the same line are "executed" at the same time. Actually, for any given group of three consecutive lines (as identified in Fig. 2), all points on these lines are executed simultaneously. In general, the number of consecutive lines to execute simultaneously corresponds to the value of $disp\Pi$. Notice that all dependencies are "satisfied" and it can be verified that the total execution time achieved by the linear schedule is the same as the one achieved by the free schedule. End of example.

The total execution time of algorithm (J, D) with a linear schedule σ_Π is

$$t(\Pi) = \max\{\sigma_\Pi(\vec{j}): \vec{j} \in J\} - \min\{\sigma_\Pi(\vec{j}): \vec{j} \in J\} + 1 = \left\lceil \frac{\max\{\Pi(\vec{j}_1 - \vec{j}_2): \vec{j}_1, \vec{j}_2 \in J\} + 1}{\min\{\Pi\vec{d}_i: \vec{d}_i \in D\}} \right\rceil \tag{2.3a}$$

or

$$t = \left\lfloor \frac{\max\{\Pi(\vec{j}_1 - \vec{j}_2): \vec{j}_1, \vec{j}_2 \in J\}}{\min\{\Pi\vec{d}_i: \vec{d}_i \in D\}} \right\rfloor + 1. \tag{2.4}$$

Because t is minimum if the argument of the floor function in (2.4) is minimized, the problem of minimizing the total execution time t can be stated as follows.

Problem 2.1 (Time optimal linear schedule problem): For algorithm (J, D) the time *optimal linear schedule problem* consists of finding a linear schedule vector $\Pi^o \in Q^{1\times n}$ such that it minimizes

$$f = \frac{\max\{\Pi(\vec{j}_1 - \vec{j}_2): \vec{j}_1, \vec{j}_2 \in J\}}{\min\{\Pi\vec{d}_i: \vec{d}_i \in D\}} \tag{2.5}$$
$$\text{subject to } \Pi\vec{d}_i > 0 \qquad \vec{d}_i \in D.$$

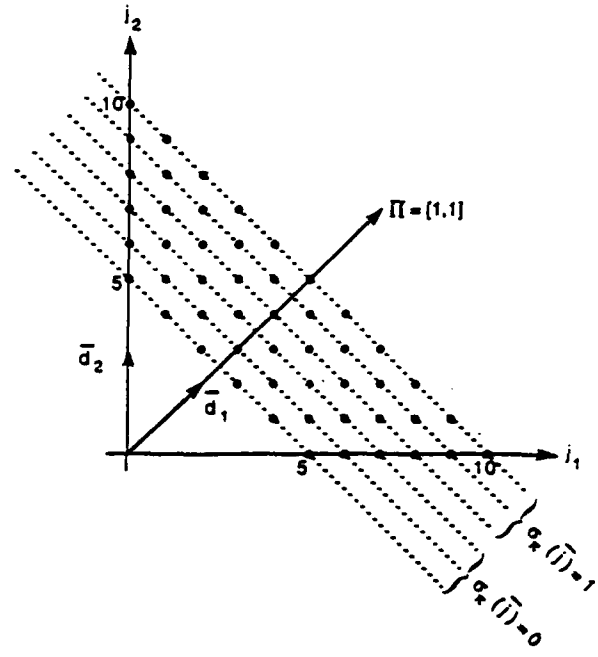


Fig. 2. The execution order by a linear schedule with $\Pi = [1, 1]$ for the algorithm of Example 2.1 when $s = 5$. Each line corresponds to a hyperplane with normal vector Π. All points on three consecutive lines can be scheduled to execute at the same time step without violating the dependency relations.

Let S denote the solution space of Problem 2.1, i.e., $S = \{\Pi: \Pi D > \vec{0}\}$. The set S includes all linear schedule vectors Π defined in Definition 2.5. Notice that if Π^o is an optimal solution of Problem 2.1, so is $\alpha\Pi^o$ for any nonzero constant α. This guarantees that the optimal solution Π^o can always be obtained such that $\Pi^o \in Z^{1\times n}$ and the greatest common divisor of the components of Π^o is equal to one. After Π^o is found, then, according to Definition 2.5, the constant c can be determined and the corresponding optimal linear schedule σ_{Π^o} can be specified completely.

As in [1], the linear schedules considered in this paper satisfy Definition 2.4 (or, equivalently, Definition 2.5). In other words, they are described by (2.3) and the condition that $disp\Pi = \min\{\Pi\vec{d}_i : \vec{d}_i \in D\} > 0$. Schedules for which $disp\Pi < \min\{\Pi\vec{d}_i: \vec{d}_i \in D\}$ are easily transformed into equivalent schedules for which Definition 2.5 is satisfied. However, there are valid schedules described by (2.3) for which $disp\Pi > \min\{\Pi\vec{d}_i: \vec{d}_i \in D\} > 0$ that are not considered in this paper. As an example, for algorithm (J, D) where $n = m = 1$, $J = \{0, 1, 3\}$, and $D = [2]$, the mapping $\sigma(j) = \lfloor ([1]j)/3 \rfloor$, $j = 0, 1, 3$, is a feasible schedule (i.e., respects the dependency). Because the denominator $disp\Pi = 3$ is greater than $\min\{\Pi\vec{d}_i: \vec{d}_i \in D\} = 1 \cdot 2 = 2$, this schedule is not a linear schedule (according to Definitions 2.4 and 2.5). However, it is easy to prove that for a mapping σ_Π described by (2.3), if there exist two index points $\vec{j}'$, $\vec{j} \in J$ such

[1] $\lfloor a \rfloor$ = the greatest integer that is less than or equal to a.

[2] $gcd(a_1, \cdots, a_n)$ = the greatest common divisor of $a_1, \cdots, a_n$.

that $(\Pi\vec{j})/disp\Pi$ is an integer and $\vec{j} + \vec{d}_i = \vec{j}'$, $\vec{d}_i \in D$, then σ_Π is a feasible schedule if and only if it meets the conditions in Definition 2.5, i.e., $disp\Pi = \min\{\Pi\vec{d}_i : \vec{d}_i \in D\}$. For example, for algorithm $(\{0,1,2,3\}, [2])$, mapping $\sigma(j) = \lfloor([1]j)/3\rfloor$, $j = 0,1,2,3$, is not a schedule because $\sigma(0) = \sigma(2)$ and the computation indexed by point 2 depends on the one indexed by 0. So, in general, for algorithms with reasonably large and dense index sets (the prevalent case for algorithms of practical interest), Definition 2.5 includes all schedules described by mappings of the form of (2.3). In fact, because loop bounds are often unknown at compile time, it is desirable to consider only those schedules that are valid regardless of the size of the algorithm. Also, for large and not necessarily dense algorithms which do not satisfy the condition mentioned above it is often possible to do simple algorithm transformations that yield an equivalent algorithm for which the condition is satisfied. In this paper, only linear schedules, as defined by Definitions 2.4 and 2.5, are considered.

The solution of Problems 2.1 is discussed in Section III. The remainder of this section contains two more definitions which are necessary for discussions in the rest of this paper.

When J is bounded, there always exist extreme points $\vec{e}_i$, $\vec{e}_k$ of J such that $\max\{\Pi(\vec{j}_1 - \vec{j}_2) : \vec{j}_1, \vec{j}_2 \in J\} = \Pi\vec{e}_i - \Pi\vec{e}_k$. To characterize this, projection vectors are defined next.

Definition 2.6 (Projection vectors): A column vector $\vec{p}$ is a *projection vector* if it is the difference of two extreme points $\vec{e}_i$, $\vec{e}_k$, of R, (i.e., $\vec{p} = \vec{e}_i - \vec{e}_k$). Hereon, the symbol q is used to denote the number of projection vectors and the set of all the projection vectors is denoted P, i.e., $P = \{\vec{p}_1, \cdots, \vec{p}_q\}$.

Example 2.4: For the algorithm of Example 2.1, its solution space is shown in Fig. 3. As it can be seen from Fig. 1, there are 12 projection vectors: $\vec{p}_1 = \vec{e}_1 - \vec{e}_2$, $\vec{p}_2 = \vec{e}_1 - \vec{e}_3$, $\vec{p}_3 = \vec{e}_1 - \vec{e}_4$, $\vec{p}_4 = \vec{e}_2 - \vec{e}_1$, $\vec{p}_5 = \vec{e}_2 - \vec{e}_3$, $\vec{p}_6 = \vec{e}_2 - \vec{e}_4$, $\vec{p}_7 = \vec{e}_3 - \vec{e}_1$, $\vec{p}_8 = \vec{e}_3 - \vec{e}_2$, $\vec{p}_9 = \vec{e}_3 - \vec{e}_4$, $\vec{p}_{10} = \vec{e}_4 - \vec{e}_1$, $\vec{p}_{11} = \vec{e}_4 - \vec{e}_2$, and $\vec{p}_{12} = \vec{e}_4 - \vec{e}_3$. End of example.

Definition 2.7 (Minimum dependence vector and maximum projection vector): For a given algorithm (J, D) and a linear schedule vector Π, a dependence vector $\vec{d}$ is called *minimum dependence vector* of Π if $\Pi\vec{d} = \min\{\Pi\vec{d}_i : \vec{d}_i \in D\} = disp\Pi$ and a projection vector $\vec{p}$ is called *maximum projection vector* of Π if $\Pi\vec{p} = \max\{\Pi\vec{p}_i : \vec{p}_i \in P\}$.

Example 2.5: Consider the algorithm of Example 2.1. The minimum dependence vector of $\Pi = [0,1]$ is $\vec{d}_1$ because $\Pi\vec{d}_1 = \min\{\Pi\vec{d}_1, \Pi\vec{d}_2\} = \min\{2,3\} = 2$ and one of its maximum projection vectors is $\vec{p}_3 = \vec{e}_1 - \vec{e}_4$. End of example.

III. Solution of the Time Optimal Linear Schedule Problem—The Bounded Index Set Case

This section considers Problem 2.1 when the index set J of the algorithm under consideration is bounded, i.e., the cardinality of J is finite, and proposes a procedure to solve this problem. While the derivation and proofs of correctness of this method are somewhat complex and long, the procedure itself is relatively simple and easy to illustrate by examples. Therefore, in Section III-A, the optimization procedure is first presented and illustrated by four examples. The formal derivation and proofs of correctness of the procedure are discussed in Section III-B and its complexity is analyzed in Section III-C.

A. Optimization Procedure

Generally speaking, Problem 2.1 is a nonlinear programming problem (in the sense that the objective function is nonlinear) with linear constraints. The solution space $S = \{\Pi : \Pi\vec{d}_i > 0, \vec{d}_i \in D\}$ contains all feasible linear schedule vectors and its cardinality is infinite. The optimization procedure proposed in this section constructs a *candidate set* C^o which is a finite subset of S and contains the optimal solution of Problem 2.1. Intuitively, the solution space S is a convex cone and it can be partitioned into a finite number of convex subcones so that the objective function in (2.5) becomes a linear fractional function in each subcone. What the optimization procedure does is to include all extreme directions of each subcone in the candidate set C^o. By the linear fractional programming theorem [14, Section 11.4], one of the extreme directions is the optimal solution of Problem 2.1.

In general, the optimal linear schedule vector Π^o is a function of the size vector $\vec{b}$ (see Example 3.4), i.e., the optimal linear schedule vector Π^o for algorithm (J, D) with a size vector $\vec{b}$ may be different from the one for the same algorithm (J, D) with a different size vector $\vec{b}'$. However, for all possible different size vectors of the same algorithm (J, D), the corresponding optimal linear schedule vectors belong to the candidate set. In this sense, the candidate set C^o is sufficient. For some algorithms (e.g., Example 3.4), it is also necessary in order to provide optimal linear schedule vectors for all possible different size vectors of the algorithm. In other words, for these algorithms, every candidate in C^o might be an optimal solution for certain given size vector $\vec{b}$ of the algorithm. For a programmed algorithm, the candidate set C^o can be constructed at compile-time. The optimal linear schedule vector Π^o can be identified either at compile-time or run-time by enumeration of all candidates in C^o, depending on whether or not the size vector $\vec{b}$ is known at compile-time.

As an informal description of the procedure, each candidate $\Pi \in C^o$ is determined by a pair consisting of a k-dimensional norm intersection of the index set J and a set of k linearly independent dependence vectors, $1 \le k \le \text{rank}(D)$. Consider a k-dimensional norm intersection $\boldsymbol{B}$ and a set of k linearly independent dependence vectors $\vec{d}_t\vec{d}_{t_1}, \cdots, \vec{d}_{t_{k-1}}$. Let $\{B_1, \cdots, B_k\}$ be a basis for $\boldsymbol{B}$ (in most cases, B_i, $i = 1, \cdots, k$, is a row vector of the index constraint matrix A), let $B = \begin{bmatrix} B_1 \\ \cdots \\ B_k \end{bmatrix}$ and let $H = [\vec{d}_t - \vec{d}_t, \vec{d}_t - \vec{d}_{t_1}, \cdots, \vec{d}_t - \vec{d}_{t_{k-1}}]$. A row vector Π is obtained as follows:

$$\Pi = \beta(\alpha_1 B_1 + \cdots + \alpha_k B_k) \tag{3.1}$$

where $\alpha_1, \cdots, \alpha_k$ satisfy equation

$$[\alpha_1, \cdots, \alpha_k]BH = \vec{0} \tag{3.2}$$

and β is a constant to be determined according to condition 2) below. For each vector Π, the following three conditions are tested:

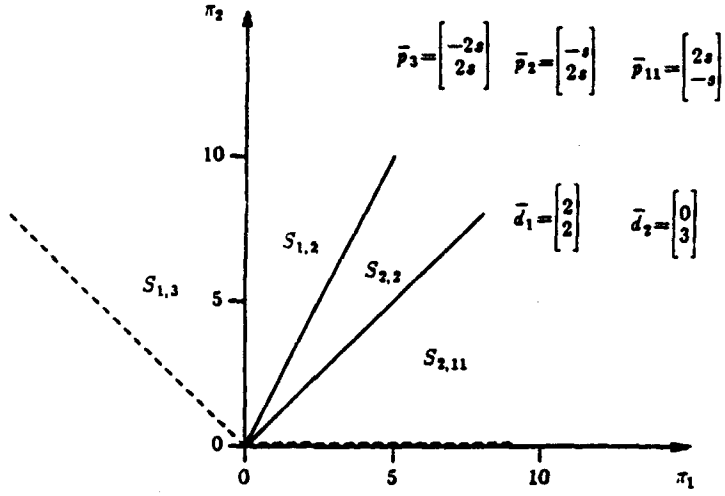


Fig. 3. The dotted lines specify the solution space S of Problem 2.1 for algorithm of Example 2.1 where $S = \{\Pi: \Pi D > \bar{0}\} = \{\Pi: \pi_1 + \pi_2 > 0, \pi_2 > 0\}$. The solid lines specify four convex subcones into which S is partitioned. For $S_{t,h}$, $t = 1, 2$, $h = 2, 3, 11$, $\bar{d}_t$ is minimum and $\bar{p}_h$ is maximum. Projection vectors are shown in Fig. 1.

1) $\text{rank}(BH) = k - 1$.
2) There exists a proper value of β such that $\Pi\bar{d}_i > 0$, $\bar{d}_i \in D$, and Π satisfies Definition 2.5.
3) The dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$ are minimum dependence vectors of Π.

If any one of the three conditions is not met, then this pair of norm intersection and dependence vectors is not good and the row vector Π is ignored (i.e., these three conditions are necessary for the optimality of Π). If all conditions are met, then the row vector Π is a candidate and included in C^o. The procedure considers every combination of each k-dimensional norm intersection with every possible distinct set of k linearly independent dependence vectors, $1 \le k \le \text{rank}(D)$, to construct the candidate set C^o. The formal description of the procedure is as follows.

Procedure 3.1 (Construction of candidate set C^o):

Input: The index constraint matrix A and the dependence matrix D of algorithm (J, D) where index set J is bounded.

Output: A finite set of candidates C^o which, for any arbitrary size vector $\bar{b}$, contains the optimal solution of Problem 2.1.

Step 1: Find all distinct norm intersections of J and set $k = 1$, $l = 1$.

Step 2: $C_k = \varnothing$.

Step 3: Pick a k-dimensional norm intersection of J and find all distinct sets of k linearly independent dependence vectors.

Step 4: Pick a set of k linearly independent dependence vectors.

Step 5: Suppose that the k dependence vectors being processed are $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$ and a basis for the k-dimensional norm intersection being processed is $\{B_1, \cdots, B_k\}$. Let $B = \begin{bmatrix} B_1 \\ \cdots \\ B_k \end{bmatrix}$ and $H = [\bar{d}_t - \bar{d}_t, \bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k-1}}]$. If $\text{rank}(BH) = k - 1$, then $\Pi_l = \beta[\alpha_1, \cdots, \alpha_k]B$ where β is a nonzero constant and $[\alpha_1, \cdots, \alpha_k]$ is a nonzero solution of equation $[\alpha_1, \cdots, \alpha_k]BH = \bar{0}$. If $\text{rank}(BH) < k - 1$, go to Step 7.

Step 6: If there does not exist a constant β such that $\Pi_l D > \bar{0}$, $\Pi_l \in Z^{1\times n}$ and the greatest common divisor of the components of Π_l is equal to one, then go to Step 7; otherwise, set the value for β and if $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$ are minimum dependence vectors of Π_l, then $C_k = C_k \cup \{\Pi_l\}$ and $l = l + 1$.

Step 7: Check if all distinct sets of k linearly independent dependence vectors have been processed. If not, pick an unprocessed set of k linearly independent dependence vectors and go to Step 5.

Step 8: Check if all distinct k-dimensional norm intersections have been processed. If not, pick an unprocessed norm intersection and go to Step 4.

Step 9: Check the value of k. If $k < \text{rank}(D)$, then $k = k + 1$, go to Step 2.

Step 10: $C^o = \bigcup_{k=1}^{\text{rank}(D)} C_k$. Stop.

To find all distinct norm intersections in Step 1, all norm spaces have to be found first by considering all possible combinations of k rows of the index constraint matrix A, $k = 1, \cdots, \min\{a, n\}$. If the k row vectors in a given combination are linearly dependent, then that combination is ignored. If they are not, the row vectors in this combination span a norm space. According to Definition 2.2, each and every norm space corresponds to a set of linearly independent rows of matrix A. Therefore, all norm spaces are found in this way and can be put in a set of norm spaces. Clearly, as mentioned before Example 2.2, there are at most $\sum_{k=1}^{\min\{a,n\}} \binom{a}{k}$ distinct norm spaces. When $k = n$, there is only one norm space which is the real space $\mathbf{R}$ and can be spanned by any set of n linearly independent rows of matrix A. Because two or more different sets of k linearly independent rows of matrix A may span the same norm space, some norm space may appear more than once in the set of norm spaces. Of course, some techniques can

TABLE I
THE NORM SPACES $N_1, \cdots, N_4$ AND NORM INTERSECTIONS $B_1, \cdots, B_4$ OF J OF THE ALGORITHM OF EXAMPLES 2.1 AND 3.1. NORM INTERSECTIONS ARE OBTAINED BY APPLYING INTERSECTION OPERATION ON TWO NORM SPACES. AS AN EXAMPLE, THE INTERSECTION OF NORM SPACES N_1 AND N_4 IS $B_1 = N_1$ WHICH IS INDICATED IN THE SECOND COLUMN AND THE LAST ROW OF THE TABLE.

	$N_1 = sp\{[-1, 0]\}$	$N_2 = sp\{[0, -1]\}$	$N_3 = sp\{[1, 1]\}$	$N_4 = sp\{[-1, 0], [1, 1]\}$
N_1	$B_1 = N_1$			
N_2	$\emptyset$	$B_2 = N_2$		
N_3	$\emptyset$	$\emptyset$	$B_3 = N_3$	
N_4	$B_1 = N_1$	$B_2 = N_2$	$B_3 = N_3$	$B_4 = N_4$

be applied to eliminate the repeated appearance of norm spaces to keep their uniqueness in the set. However, this repeated appearance in the set does not affect the correctness of the final result of Procedure 3.1.

All norm intersections can be found by intersecting each norm space with every other norm space. There are at most $(\sum_{k=1}^{\min\{a,n\}} \binom{a}{k})(\sum_{k=1}^{\min\{a,n\}} \binom{a}{k} - 1)$ norm intersections. As explained later after Example 3.4, for many algorithms, each and every norm intersection is equal to some norm space and therefore, there is no need to put effort in finding norm intersections. To find a basis for the norm intersection B of two norm spaces $sp\{A_{t_1}, \cdots, A_{t_{k_1}}\}$ and $sp\{A_{h_1}, \cdots, A_{h_{k_2}}\}$ (needed in Step 5) requires the solution of the following equation:

$$[\delta_1, \cdots, \delta_{k_1}, \delta_{k_1+1}, \cdots, \delta_{k_1+k_2}] \begin{bmatrix} A_{t_1} \\ \cdots \\ A_{t_{k_1}} \\ A_{h_1} \\ \cdots \\ A_{h_{k_2}} \end{bmatrix} = \bar{0}. \qquad (3.3)$$

If there are k linearly independent solutions $\Delta_i = [\delta_{i1}, \delta_{i2}, \cdots, \delta_{i(k_1+k_2)}]$, $i = 1, \cdots, k$ for (3.3), then the dimension of B is k and a basis for B is

$$[\delta_{i1} \cdots \delta_{ik_1}] \begin{bmatrix} A_{t_1} \\ \cdots \\ A_{t_{k_1}} \end{bmatrix} \qquad i = 1, \cdots, k.$$

The following examples are used to show how the candidate set C^o is constructed according to Procedure 3.1 and illustrate different aspects of it. Detailed discussion of different aspects of Procedure 3.1 is given after these examples.

Example 3.1: Procedure 3.1 is now applied to find the optimal linear schedule vector Π^o for the algorithm of Example 2.1. There are 8 boundary surfaces, four of which correspond to the four boundary lines of J and the remaining four correspond to the four extreme points of J. There are four distinct norm spaces N_i, $i = 1, \cdots, 4$ and four distinct norm intersections B_i, $i = 1, \cdots, 4$, which are listed in Table I. The derivation of these norm intersections is shown in Table I. Notice that the set of two linearly independent rows $\{[-1, 0], [0, -1]\}$ can span a norm space N_5 which is equal to N_4. Norm space N_5 is removed from the set of norm spaces because it is known that there is only one n-dimensional norm space for any algorithms. In Step 1, the norm intersections are found, i.e., $B_1 = sp\{[-1, 0]\}$, $B_2 = sp\{[0, -1]\}$, $B_3 = sp\{[1, 1]\}$, and $B_4 = sp\{[-1, 0], [1, 1]\}$ and k is assigned the value one. According to Steps 3 and 4, each one-dimensional norm intersection (i.e., B_1, B_2, and B_3) is considered together with each of the dependence vectors (i.e., $\bar{d}_1$ and $\bar{d}_2$). According to Step 5 and because for all pairs $(B_i, \bar{d}_l)$, $i = 1, 2, 3$, $l = 1, 2$, $H = \bar{0}$, $\text{rank}(BH) = 0 = k - 1$, the following distinct vectors are generated: $\Pi_1 = \beta_1[-1, 0]$, $\Pi_2 = \beta_2[0, -1]$, and $\Pi_3 = \beta_3[1, 1]$. For Step 6, Π_1 is not feasible because $\Pi_1 \bar{d}_2 = 0$ regardless of the value of β_1. If $\beta_2 = -1$ and $\beta_3 = 1$, then $\Pi_2 D > \bar{0}$ and $\Pi_3 D > \bar{0}$, Π_2, $\Pi_3 \in Z^{1\times 2}$ and the greatest common divisors of the entries of both Π_2 and Π_3 are unity; $\bar{d}_1$ is the minimum dependence vector of Π_2 and $\bar{d}_2$ is the minimum dependence vector of Π_3 (i.e., Π_2 and Π_3 correspond to the pairs $(B_2, \bar{d}_1)$ and $(B_3, \bar{d}_2)$, respectively.) So, $C_1 = \{\Pi_2, \Pi_3\}$. Let $k = 2$, there is only one two-dimensional norm intersection ($B_4 = R^2$) and only one set of two linearly independent dependence vectors $\{\bar{d}_1, \bar{d}_2\}$. According to Step 5, $B = \begin{bmatrix} -1 & 0 \\ 1 & 1 \end{bmatrix}$, $H = \begin{bmatrix} 2 & 0 \\ -1 & 0 \end{bmatrix}$, and $\text{rank}(BH) = 1 = k - 1$. By equation $[\alpha_1, \alpha_2]BH = 0$, $[\alpha_1, \alpha_2] = [1, 2]$ and the corresponding row vector $\Pi_4 = \beta_4[1, 2]$. Similarly, if $\beta_4 = 1$, conditions in Step 6 are satisfied and $\bar{d}_1$ and $\bar{d}_2$ are minimum dependence vectors of Π_4. Therefore, $C_2 = \{\Pi_4\}$ and $C^o = C_1 \cup C_2 = \{\Pi_2, \Pi_3, \Pi_4\}$. C^o is a finite subset of the solution space S which is shown in Fig. 3. The candidate with shortest execution time in C^o is the optimal solution of Problem 2.1, i.e., the optimal linear schedule vector for this algorithm. The total execution time by each candidate in C^o is evaluated according to (2.3a) as follows:

$$t(\Pi_2) = \left\lceil \frac{[0, 1]\left([0, 2s]^T - [s, 0]^T\right) + 1}{2} \right\rceil = \left\lceil \frac{2s + 1}{2} \right\rceil$$

$$t(\Pi_3) = \left\lceil \frac{[1, 1]\left([2s, 0]^T - [s, 0]^T\right) + 1}{3} \right\rceil = \left\lceil \frac{s + 1}{3} \right\rceil$$

$$t(\Pi_4) = \left\lceil \frac{[1, 2]\left([0, 2s]^T - [s, 0]^T\right) + 1}{6} \right\rceil = \left\lceil \frac{3s + 1}{6} \right\rceil.$$

Clearly, Π_3 is of the shortest execution time. Therefore, the optimal linear schedule vector for this algorithm is Π_3, i.e., $\Pi^o = \Pi_3$. In this example, because there is only one variable s in the size vector $\bar{b}$, the optimal linear schedule vector is independent of the size vector. End of example.

Example 3.2: Consider the algorithm (J, D) with $D = [\bar{d}_1, \bar{d}_2, \bar{d}_3]$, where $\bar{d}_1 = [1, -3]^T$, $\bar{d}_2 = [2, 4]^T$, and $\bar{d}_3 = [2, 0]^T$. The index set J is the same as the algorithm of Example 2.1 and shown in Fig. 1. Because the index set of

TABLE II

NORM SPACES AND NORM INTERSECTIONS OF EXAMPLE 3.3. THE DERIVATION OF THE NORM INTERSECTIONS IS AS DESCRIBED IN TABLE I. NOTICE THAT B_i, $i = 5, 6, 7$, ARE DIFFERENT FROM ALL THE NORM SPACES.

	$N_1 = sp\{[-1,0,1]\}$	$N_2 = sp\{[-1,1,0]\}$	$N_3 = sp\{[1,0,0]\}$	$N_4 = sp\{[1,-1,-1]\}$	$N_5 = sp\{[-1,0,1], [-1,1,0]\}$	$N_6 = sp\{[-1,0,1], [1,0,0]\}$	$N_7 = sp\{[-1,0,1], [1,-1,-1]\}$	$N_8 = sp\{[-1,1,0], [1,0,0]\}$	$N_9 = sp\{[-1,1,0], [1,-1,-1]\}$	$N_{10} = sp\{[1,0,0], [1,-1,-1]\}$	$N_{11} = sp\{[-1,0,1], [-1,1,0], [1,0,0]\}$
N_1	$B_1 = N_1$										
N_2	∅	$B_2 = N_2$									
N_3	∅	∅	$B_3 = N_3$								
N_4	∅	∅	∅	$B_4 = N_4$							
N_5	B_1	B_2	∅	∅	$B_5 = N_5$						
N_6	B_1	∅	B_3	∅	B_1	$B_6 = N_6$					
N_7	B_1	∅	∅	B_4	B_1	B_1	$B_{10} = N_7$				
N_8	∅	B_2	B_3	∅	B_2	B_3	$B_5 = sp\{[0,1,0]\}$	$B_{11} = N_8$			
N_9	∅	B_2	∅	B_4	B_2	$B_6 = sp\{[0,0,1]\}$	B_4	B_2	$B_{12} = N_9$		
N_{10}	∅	∅	B_3	B_4	$B_7 = sp\{[0,1,1]\}$	B_3	B_4	B_7	B_4	$B_{13} = N_{10}$	
N_{11}	B_1	B_2	B_3	B_4	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	$B_{14} = N_{11}$

this algorithm is the same as the one of the algorithm of Example 3.1, the norm intersections of J are the same which are shown in Table I. For $k = 1$, similarly to Example 3.1, all pairs $(B_i, \bar{d}_l)$, $i, l = 1, 2, 3$, are considered. According to Step 5, three distinct row vectors are generated: $\Pi_1 = \beta_1[-1, 0]$, $\Pi_2 = \beta_2[0, -1]$, and $\Pi_3 = \beta_3[1, 1]$. For Step 6, Π_2 and Π_3 are not feasible because $\Pi_2\bar{d}_3 = 0$ regardless of the value of β_2 and, $\Pi_3\bar{d}_1 < 0$ if $\beta_3 > 0$, $\Pi_3\bar{d}_2 < 0$ if $\beta_3 < 0$. If $\beta_1 = -1$, $\Pi_1 D > \bar{0}$, $\Pi_1 \in Z^{1\times 2}$ and the greatest common divisor of the components of Π_1 is unity. The minimum dependence vector of Π_1 is $\bar{d}_1$ [i.e., Π_1 corresponds to the pair $(B_1, \bar{d}_1)$]. So, $C_1 = \{\Pi_1 = [1, 0]\}$. For $k = 2$, there is only one two-dimensional norm intersection B_4 and there are three distinct sets of two linearly independent dependence vectors: $\{\bar{d}_1, \bar{d}_2\}$, $\{\bar{d}_1, \bar{d}_3\}$, and $\{\bar{d}_2, \bar{d}_3\}$. Thus, there are three pairs of a two-dimensional norm intersection and a set of two linearly independent dependence vectors: $(B_4, \{\bar{d}_1, \bar{d}_2\})$, $(B_4, \{\bar{d}_1, \bar{d}_3\})$, and $(B_4, \{\bar{d}_2, \bar{d}_3\})$. According to Step 5, three distinct row vectors are generated: $\Pi_4 = \beta_4[7, -1]$, $\Pi_5 = \beta_5[3, -1]$, and $\Pi_6 = \beta_6[1, 0]$. For Step 6, Π_4, Π_5, and Π_6 are all feasible. However, Π_5 should not be included in C_2 because Π_5 corresponds to the pair $(B_4, \{\bar{d}_1, \bar{d}_3\})$ and $\bar{d}_1, \bar{d}_3$ are not its minimum dependence vectors. For the same reason, Π_6 should not be included in C_2. If $\beta_4 = 1$, $\Pi_4 D > \bar{0}$, $\Pi_4 \in Z^{1\times 2}$ and the greatest common divisor of the components of Π_4 is unity. Vector Π_4 corresponds to the pair $(B_4, \{\bar{d}_1, \bar{d}_2\})$ and $\bar{d}_1, \bar{d}_2$ are minimum dependence vectors of Π_4. So, Π_4 should be included in C_2. Therefore, $C_2 = \{\Pi_4 = [7, -1]\}$ and $C^o = \{\Pi_1, \Pi_4\}$. To find the optimal linear schedule, it is necessary to compare the execution times for all candidates in C^o. The total execution time by each candidate is evaluated as follows:

$$t(\Pi_1) = \left\lceil \frac{[1,0]\left([2s,0]^T - [0,s]^T\right) + 1}{1} \right\rceil = 2s + 1$$

$$t(\Pi_4) = \left\lceil \frac{[7,-1]\left([2s,0]^T - [0,2s]^T\right) + 1}{10} \right\rceil = \left\lceil \frac{16s+1}{10} \right\rceil$$

Clearly, Π_4 has the shortest execution time. Therefore, $\Pi^o = \Pi_4 = [7, -1]$. End of example.

Example 3.3: Consider the algorithm (J, D) where $J = \{\vec{j} : A\vec{j} \le \bar{b}, \vec{j} \in Z^3\}$,

$$A = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & -1 \end{bmatrix}, \qquad \bar{b} = \begin{bmatrix} 0 \\ 0 \\ s \\ 0 \end{bmatrix},$$

and dependence matrix

$$D = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

The index set J is shown in Fig. 4. Pictorially, J is surrounded by four planes described by equations $x_1 - x_3 = 0$, $x_1 - x_2 = 0$, $x_1 = s$, and $-x_1 + x_2 + x_3 = 0$. For Step 1, there are 11 distinct norm spaces N_i, $i = 1, \cdots, 11$, listed in Table II. The corresponding boundary surfaces F_i of N_1, $i = 1, \cdots, 11$ are indicated in Fig. 4. There are four two-dimensional boundary surfaces (planes) $F_1, \cdots, F_4$, six one-dimensional boundary surfaces (lines) $F_5, \cdots, F_{10}$ and one zero-dimensional boundary surface F_{11}. The derivations of all norm intersections are shown in Table II. There are 14 distinct norm intersections, i.e., $B_1 = sp\{[-1, 0, 1]\}$, $B_2 = sp\{[-1, 1, 0]\}$, $B_3 = sp\{[1, 0, 0]\}$, $B_4 = sp\{[1, -1, -1]\}$, $B_5 = sp\{[0, 1, 0]\}$, $B_6 = sp\{[0, 0, 1]\}$, $B_7 = sp\{[0, 1, 1]\}$, $B_8 = sp\{[-1, 0, 1], [-1, 1, 0]\}$, $B_9 = sp\{[-1, 0, 1],$

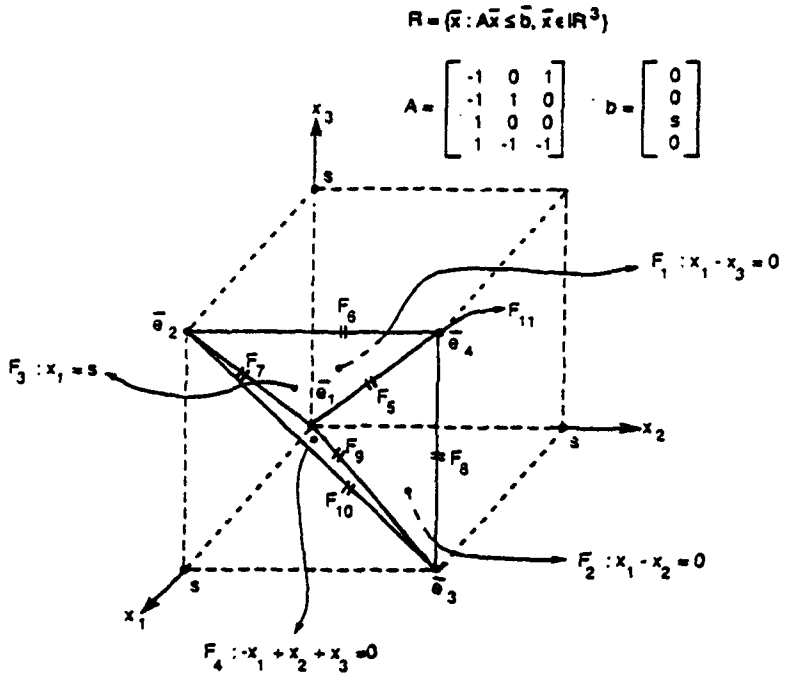


Fig. 4. The 4-face polyhedron inside the cube is the index set of the algorithm of Example 3.3. It has four extreme points $\bar{e}_1, \cdots, \bar{e}_4$, four two-dimensional boundary surfaces $F_1, \cdots, F_4$ and six one-dimensional boundary surfaces $F_5, \cdots, F_{10}$. F_i, is associated with the norm space N_i, $i = 1, \cdots, 11$, listed in Table II.

$[1,0,0]\}$, $B_{10} = sp\{[-1,0,1],[1,-1,-1]\}$, $B_{11} = sp\{[-1, 1.0],[1,0,0]\}$, $B_{12} = sp\{[-1,1,0],[1,-1,-1]\}$, $B_{13} = sp\{[1,0,0],[1,-1,-1]\}$, and $B_{14} = sp\{[-1,0,1],[-1,1,0],[1,0,0]\}$. Notice that B_5, B_6, and B_7 are no longer norm spaces of J. So, in general, it is necessary to identify all norm intersections of J instead of the norm space of J. As an example, if a candidate Π belongs to B_5, the intersection of norm spaces N_7 and N_8, then it is perpendicular to both F_7 and F_8 which are associated to N_7 and N_8, respectively. Table III lists all row vectors and all candidates in C^o generated by Procedure 3.1 and their corresponding pairs of a k-dimensional norm intersection and a set of k linearly independent dependence vectors. Each row of Table III corresponds to a row vector and its corresponding pair of a k-dimensional norm intersection and a set of k linearly independent dependence vectors. Column 1 indicates the value of k. Column 2 lists the corresponding norm intersections. Column 3 lists the corresponding sets of dependence vectors. Column 4 lists the corresponding row vectors obtained according to Step 5 in Procedure 3.1 if $\text{rank}(BH) = k$, or "NOT OK" if $\text{rank}(BH) < k-1$ (B and H are defined in Step 5 of Procedure 3.1). Column 5 tests the feasibility of the corresponding row vector Π. The case where Π is not feasible is indicated by "NO." If Π is feasible, then it indicates the value of constant β_i (i.e., β in Step 5 of Procedure 3.1) such that $\Pi D > \bar{0}$, $\Pi \in Z^{1\times n}$ and the greatest common divisor of the components of Π is unity. Column 6 tests if the dependence vectors in column 2 are minimum dependence vectors of Π. Column 7 gives the execution time of the candidates. For $k = 1$, there should be 21 distinct pairs of a one-dimensional norm intersection and a set of one dependence vector because there are seven one-dimensional norm intersections and three dependence vectors. However, only seven such pairs are listed in the first seven rows of Table III because, according to Step 5, only seven distinct row vectors are generated by these 21 pairs. For $k = 2$, there are 18 distinct pairs of a two-dimensional norm intersection and a set of two linearly independent dependence vectors because there are six two-dimensional norm intersections and three distinct sets of two linearly independent dependence vectors. All these pairs are listed in Table III. For $k = 3$, there is only one such pair corresponding to the last row of Table III. There are only two distinct candidates in C^o, i.e., $C^o = \{[0,1,0],[0,1,1]\}$. Because the linear schedule vector $[0,1,0]$ has shorter execution time, $\Pi^o = [0,1,0]$ and the total execution time is $s+1$. End of example.

Example 3.4: Consider the algorithm with index set $J = \{\vec{j} : A\vec{j} \le \vec{b}, \vec{j} \in Z^2\}$ and dependence matrix

$$D = \begin{bmatrix} 2 & 0 \\ 2 & 3 \end{bmatrix}$$

where

$$A = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$\vec{b} = \begin{bmatrix} 0 \\ 0 \\ s_1 \\ s_2 \end{bmatrix}.$$

The index set J is shown in Fig. 5 and the dependence matrix is the same as the algorithm of Example 2.1. According to Procedure 3.1, the candidate set C^o is constructed as $C^o =$

TABLE III

DERIVATION OF ALL CANDIDATES OF ALGORITHM OF EXAMPLE 3.3. COLUMN 1 INDICATES THE VALUE OF k; COLUMN 2 LISTS THE CORRESPONDING NORM INTERSECTIONS; COLUMN 3 LISTS THE CORRESPONDING DEPENDENCE VECTORS; COLUMN 4 LISTS THE CORRESPONDING ROW VECTORS OBTAINED ACCORDING TO STEP 5 IN PROCEDURE 3.1; COLUMN 5 TESTS THE FEASIBILITY OF THE ROW VECTOR; COLUMN 6 TESTS IF THE DEPENDENCE VECTORS IN COLUMN 3 ARE MINIMUM AND COLUMN 7 LISTS THE EXECUTION TIMES OF THE LINEAR SCHEDULE VECTORS IN COLUMN 4.

k	Norm Intersection	$\bar{d}$-vector	Row Vector	Feasible?	Minimum?	Execution Time
1	$\mathbf{B}_1=sp\{[-1,0,1]\}$	$\bar{d}_1$	$\Pi_1=\beta_1[-1,0,1]$	No		
1	$\mathbf{B}_2=sp\{[-1,1,0]\}$	$\bar{d}_1$	$\Pi_2=\beta_2[-1,1,0]$	No		
1	$\mathbf{B}_3=sp\{[1,0,0]\}$	$\bar{d}_2$	$\Pi_3=\beta_3[1,0,0]$	No		
1	$\mathbf{B}_4=sp\{[1,-1,-1]\}$	$\bar{d}_3$	$\Pi_4=\beta_4[1,-1,-1]$	No		
1	$\mathbf{B}_5=sp\{[0,1,0]\}$	$\bar{d}_1$	$\Pi_5=\beta_5[0,1,0]$	$\beta_5=1$	Yes	$s+1$
1	$\mathbf{B}_6=sp\{[0,0,1]\}$	$\bar{d}_2$	$\Pi_6=\beta_6[0,0,1]$	No		
1	$\mathbf{B}_7=sp\{[0,1,1]\}$	$\bar{d}_2$	$\Pi_7=\beta_7[0,1,1]$	$\beta_7=1$	Yes	$2s+1$
2	$\mathbf{B}_8=sp\{[-1,0,1],$ $[-1,1,0]\}$	$\bar{d}_1,\bar{d}_2$	$\Pi_8=\beta_8[-1,0,1]$	No		
2	$\mathbf{B}_9$	$\bar{d}_1,\bar{d}_3$	$\Pi_9=\beta_9[-1,1,0]$	No		
2	$\mathbf{B}_9$	$\bar{d}_2,\bar{d}_3$	$\Pi_{10}=\beta_{10}[0,1,-1]$	No		
2	$\mathbf{B}_9=sp\{[-1,0,1],$ $[1,0,0]\}$	$\bar{d}_1,\bar{d}_2$	$\Pi_{11}=\beta_{11}[-1,0,1]$	No		
2	$\mathbf{B}_9$	$\bar{d}_1,\bar{d}_3$	$\Pi_{12}=\beta_{12}[1,0,0]$	No		
2	$\mathbf{B}_9$	$\bar{d}_2,\bar{d}_3$	$\Pi_{13}=\beta_{13}[0,0,1]$	No		
2	$\mathbf{B}_{10}=sp\{[-1,0,1],$ $[1,-1,-1]\}$	$\bar{d}_1,\bar{d}_2$	Not OK			
2	$\mathbf{B}_{10}$	$\bar{d}_1,\bar{d}_3$	$\Pi_{15}=\beta_{15}[0,1,0]$	$\beta_{15}=1$	Yes	$s+1$
2	$\mathbf{B}_{10}$	$\bar{d}_2,\bar{d}_3$	$\Pi_{16}=\beta_{16}[0,1,0]$	$\beta_{16}=1$	Yes	$s+1$
2	$\mathbf{B}_{11}=sp\{[-1,1,0],$ $[1,0,0]\}$	$\bar{d}_1,\bar{d}_2$	$\Pi_{17}=\beta_{17}[0,1,0]$	$\beta_{17}=1$	Yes	$s+1$
2	$\mathbf{B}_{11}$	$\bar{d}_1,\bar{d}_3$	Not OK			
2	$\mathbf{B}_{11}$	$\bar{d}_2,\bar{d}_3$	$\Pi_{19}=\beta_{19}[0,1,0]$	$\beta_{19}=1$	Yes	$s+1$
2	$\mathbf{B}_{12}=sp\{[-1,1,0],$ $[1,-1,-1]\}$	$\bar{d}_1,\bar{d}_2$	$\Pi_{20}=\beta_{20}[1,-1,-1]$	No		
2	$\mathbf{B}_{12}$	$\bar{d}_1,\bar{d}_3$	$\Pi_{21}=\beta_{21}[-1,1,0]$	No		
2	$\mathbf{B}_{12}$	$\bar{d}_2,\bar{d}_3$	$\Pi_{22}=\beta_{22}[0,0,1]$	No		
2	$\mathbf{B}_{13}=sp\{[1,0,0],$ $[1,-1,-1]\}$	$\bar{d}_1,\bar{d}_2$	$\Pi_{23}=\beta_{23}[1,-1,-1]$	No		
2	$\mathbf{B}_{13}$	$\bar{d}_1,\bar{d}_3$	$\Pi_{24}=\beta_{24}[1,0,0]$	No		
2	$\mathbf{B}_{13}$	$\bar{d}_2,\bar{d}_3$	$\Pi_{25}=\beta_{25}[0,1,1]$	$\beta_{25}=1$	Yes	$2s+1$
3	$\mathbf{B}_{14}=R^3$	$\bar{d}_1,\bar{d}_2,\bar{d}_3$	$\Pi_{26}=\beta_{26}[0,1,0]$	$\beta_{26}=1$	Yes	$s+1$

$\{[0,1],[1,2]\}$. The total execution time by each candidate is evaluated as follows.

$$t([0,1]) = \left\lceil \frac{[0,1]\left([0,s_2]^T - [0,0]^T\right)+1}{2} \right\rceil = \left\lceil \frac{s_2+1}{2} \right\rceil$$

$$t([1,2]) = \left\lceil \frac{[1,2]\left([s_1,s_2]^T - [0,0]^T\right)+1}{6} \right\rceil = \left\lceil \frac{s_1+2s_2+1}{6} \right\rceil.$$

Clearly, the optimal linear schedule depends on variables s_1 and s_2, i.e., size vector $\bar{b}$. If $s_2 < s_1 - 2$, then $(s_2+1)/2 < (s_1+2s_2+1)/6$ and $[0,1]$ is the optimal solution. Similarly, when $s_2 > s_1 - 2$, vector $[1,2]$ is the optimal solution. Therefore, for this algorithm, the candidate set C^o is necessary, i.e., every candidate is an optimal solution for some size vector. End of example.

From the discussion of the above examples it is possible to get additional insights into the candidates identified by Procedure 3.1. Let $\mathbf{B} = sp\{B_1, \cdots, B_k\}$ be a k-dimensional norm intersection of the norm spaces associated with boundary

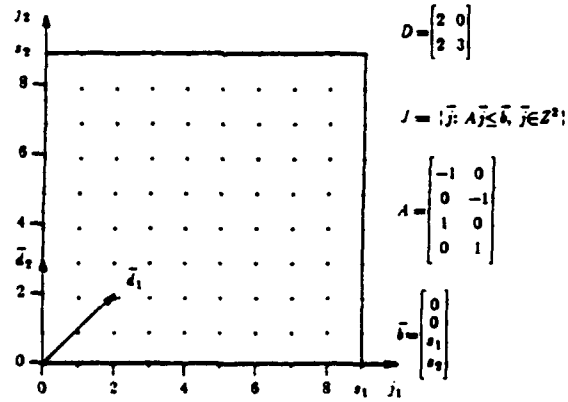


Fig. 5. The index set of algorithm of Example 3.4. The optimal linear schedule depends on size vector $\bar{b}$ because it has two variables s_1 and s_2.

surfaces F_1 and F_2, respectively. Intuitively, if a candidate $\Pi \in C^o$ is determined by $\mathbf{B}$ and a set of k linearly independent dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$, then Π is perpendicular to both boundary surfaces F_1 and F_2 and $\Pi\bar{d}_t = \Pi\bar{d}_{t_1} = \cdots =$

$\Pi\bar{d}_{t_{k-1}}$, i.e., Π satisfies the following equation

$$\begin{cases} \Pi(\bar{d}_t - \bar{d}_t) = 0 \\ \Pi(\bar{d}_t - \bar{d}_{t_1}) = 0 \\ \cdots \\ \Pi(\bar{d}_t - \bar{d}_{t_{k-1}}) = 0. \end{cases} \tag{3.4}$$

For example, in Example 3.1, linear schedule vector Π_3 is determined by the pair $(B_3, \bar{d}_2)$. As it can be seen in Fig. 1, the norm intersection B_3 can be thought as the intersection of the norm spaces associated with $F_3 = \text{surf}\{A_3\}$ and $F_5 = \text{surf}\{A_1, A_4\}$. Surface F_3 is the line through points $\bar{e}_1$ and $\bar{e}_4$ and F_5 is the extreme point $\bar{e}_2$ shown in Fig. 1. It can be verified that Π_3 is perpendicular to both surfaces F_3 and F_5 and satisfies equation $\Pi_3(\bar{d}_2 - \bar{d}_2) = 0$. Vector Π_4 is determined by the pair $(B_4, \{\bar{d}_1, \bar{d}_3\})$ where B_4 is the intersection of the norm spaces associated with any two extreme points of J shown in Fig. 1. It can be verified that Π_4 is perpendicular to any extreme points and satisfies equations $\Pi_4(\bar{d}_1 - \bar{d}_1) = 0$ and $\Pi_4(\bar{d}_1 - \bar{d}_2) = 0$.

The above examples also illustrate the following different aspects of Procedure 3.1. First, Examples 3.1 and 3.2 show that, to find the optimal linear schedule vector, not only the dependence vectors should be considered, but also the boundary surfaces, or the shape of the index set must be taken into account. In Example 3.1, the optimal linear schedule vector Π^o is determined by boundary surfaces $F_3 = \text{surf}\{A_3\}$ and $F_5 = \text{surf}\{A_1, A_4\}$, i.e., Π^o is perpendicular to both F_3 and F_5. In contrast, the optimal linear schedule vector Π^o for the algorithm of Example 3.2 is determined by (3.4) with two linearly independent dependence vectors $\bar{d}_1$ and $\bar{d}_2$. Second, for most algorithms (e.g., the algorithm of Example 3.1) each norm intersection is still one of the norm spaces of J. It is shown in Section V that this is true for the algorithms whose index sets are shaped as a hyperparallelepiped. In this case, there is no need to find any norm intersections. However, in general, some norm intersections are different from all norm spaces as indicated by Example 3.3 (e.g., B_5 is different from all norm spaces). Third, Example 3.4 indicates that, the optimal linear schedule vector Π^o may depend on the size vector $\bar{b}$ and each candidate in C^o is an optimal linear schedule vector for some instance of the algorithm. Finally, when $k = 1$, (3.2) is trivial because $H = \bar{0}$. The corresponding row vector Π is solely determined by (3.1). When $k = n$, the n-dimensional norm intersection is simply $\mathbf{R}^n$ and (3.1) is trivial. In this case, the corresponding row vector can be determined solely by (3.4).

According to the observations above, it is clear that for two-dimensional algorithms (Examples 3.1, 3.2, and 3.4), because all norm intersections are equal to some norm spaces, there is no need to find norm intersections. Therefore, what Procedure 3.1 does for two-dimensional algorithms is, for each and every boundary line, to put the vector perpendicular to that boundary line in the candidate set if this vector is feasible. In addition, Procedure 3.1 considers all distinct sets of two linearly independent dependence vectors $\{\bar{d}_i, \bar{d}_l\}$ and puts the vector Π such that $\Pi(\bar{d}_i - \bar{d}_l) = 0$ in the candidate set if Π is feasible.

B. Correctness of the Procedure

In this subsection, it is shown that Π^o, the optimal solution of Problem 2.1, belongs to the candidate set C^o constructed by Procedure 3.1. This fact is stated as Theorem 3.1 and is proven by using six lemmas. All the proofs of these lemmas are provided in the Appendix. However, an example is presented to illustrate the ideas behind these proofs.

Theorem 3.1: The optimal solution Π^o of Problem 2.1 belongs to C^o, the candidate set constructed by Procedure 3.1, i.e., $\Pi^o \in C^o$.

The basic idea behind the proof of Theorem 3.1 is as follows. First, it is shown that the solution space S is a convex cone (Lemma 1) that can be partitioned into convex subcones S_{th} where $\bar{d}_t$ is the minimum dependency and $\bar{p}_h$ is the maximum projection vector (Lemmas 2 and 3). In other words, for any arbitrary Π in S_{th}, $\max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\} = \Pi\bar{p}_h$ and $\min\{\Pi\bar{d}_i : \bar{d}_i \in D\} = \Pi\bar{d}_t$ which implies that the objective function described by (2.5) simply becomes a linear fractional function of Π as follows (Lemma 2):

$$f = \frac{\Pi\bar{p}_h}{\Pi\bar{d}_t} \qquad \Pi \in S_{th}. \tag{3.5}$$

Each direction [14, p. 55] in the convex cone S corresponds a feasible solution of Problem 2.1. Because for every $\Pi \in S$, $f(\lambda\Pi) = f(\Pi)$ for any nonzero constant λ, f is a function of directions in the cone instead of the points, i.e., for all points on the same direction except the origin, the objective function takes the same value (Lemma 1). According to [14, Section 11.4], Problem 2.1 is a *linear fractional program* in each convex subcone S_{th}, and the optimal direction of f over S_{th} is one of the extreme directions of the subcone S_{th} (Lemma 4). Therefore, one of the extreme directions of S_{th}, $t = 1, \cdots, m$ and $h = 1, \cdots, q$, must be the optimal solution Π^o. Finally, it is shown that all such extreme directions are included in the candidate set C^o constructed by Procedure 3.1 (Lemmas 5, 6), i.e., each candidate in C^o corresponds to an extreme direction of convex subcone S_{th} for some t and h and vice versa. Therefore, the candidate in C^o with the shortest execution time is an optimal solution of Problem 2.1.

Lemma 1: The solution space S is a convex cone and the objective function $f = (\max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\})/(\min\{\Pi\bar{d}_i : \bar{d}_i \in D\})$ is a function of the directions in S.

Lemma 2: Let

$$S_{th} = \bigcap_{\substack{i=1 \\ i \neq t}}^{m} \{\Pi : \Pi\bar{d}_t \leq \Pi\bar{d}_i\} \bigcap_{\substack{i=1 \\ i \neq h}}^{q} \{\Pi : \Pi\bar{p}_h \geq \Pi\bar{p}_i\} \cap S \tag{3.6}$$

$$= \{\Pi : \Pi(\bar{d}_t - \bar{d}_i) \leq 0, \Pi(\bar{p}_k - \bar{p}_h) \leq 0, \quad i = 1, \cdots, m, \ k = 1, \cdots, q, \Pi \in S\}. \tag{3.7}$$

If S_{th} is not empty, then S_{th} is a convex cone where the objective function can be reexpressed as

$$f = \frac{\max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}}{\min\{\Pi\bar{d}_i : \bar{d}_i \in D\}} = \frac{\Pi\bar{p}_h}{\Pi\bar{d}_t}.$$

Lemma 3: $S = \bigcup_{t=1}^{m} \bigcup_{h=1}^{q} S_{th}$.

Lemma 4: An optimal direction for the objective function $f = (\Pi \bar{p}_h)/(\Pi \bar{d}_t)$ over S_{th} is one of the extreme directions of S_{th}.

Lemma 5: Let $\Pi^e \in S_{th}$. Then Π^e is an extreme direction of S_{th} if and only if there exists a set of $n-1$ linearly independent vectors $\bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}, \bar{p}_h - \bar{p}_{h_1}, \cdots, \bar{p}_h - \bar{p}_{h_{n-k'}}$ such that Π^e satisfies the following equations:

$$\text{(a)} \begin{cases} \Pi(\bar{d}_t - \bar{d}_t) = 0 \\ \Pi(\bar{d}_t - \bar{d}_{t_1}) = 0 \\ \cdots \\ \Pi\left(\bar{d}_t - \bar{d}_{t_{k'-1}}\right) = 0 \end{cases} \qquad \text{(b)} \begin{cases} \Pi(\bar{p}_h - \bar{p}_{h_1}) = 0 \\ \cdots \\ \Pi\left(\bar{p}_h - \bar{p}_{h_{n-k'}}\right) = 0 \end{cases} \tag{3.8}$$

where $1 \le k' \le \text{rank}(D)$.

Lemma 6: Let $\Pi^e \in S_{th}$ be an extreme direction satisfying (3.8). Then the following statements are true.

a) $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k'-1}}$ are linearly independent and are minimum dependence vectors of Π^e.

b) There exists a k-dimensional norm intersection $B = sp\{B_1, \cdots, B_k\}$ such that $\Pi^e \in B$ and $k \le k'$, i.e., for some constants α_i, $i = 1, \cdots, k$

$$\Pi^e = \sum_{i=1}^{k} \alpha_i B_i. \tag{3.9}$$

c) Let $B = \begin{bmatrix} B_1 \\ \cdots \\ B_k \end{bmatrix}$ and $H' = [\bar{d}_t - \bar{d}_t, \bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}]$. Then $\text{rank}(BH') = k - 1$.

d) Without loss of generality, let $H = [\bar{d}_t - \bar{d}_t, \bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k-1}}]$ be such that $\text{rank}(BH) = k - 1$. Then Π^e can be obtained from (3.9) and the equation

$$[\alpha_1, \cdots, \alpha_k]BH = \bar{0}. \tag{3.10}$$

Lemma 6 simply states (3.8) is equivalent to (3.9) and (3.10), i.e., Π^e is a solution of (3.8) if and only if it is a solution of (3.9) and (3.10). This provides another way to find all extreme directions because in practice, it is not easy to identify all projection vectors, especially, when the number of the extreme points is large. All extreme directions can be more easily obtained by (3.9) and (3.10).

Example 3.5: The algorithm of Example 2.1 is used to illustrate Lemmas 1–6. The solution space of Problem 2.1 for this algorithm is shown in Fig. 3. Lemmas 1–4 are illustrated as follows. The solution space S is a convex cone $\{\Pi: \Pi\bar{d}_1 > 0, \Pi\bar{d}_2 > 0\} = \{\Pi: 2\pi_1 + 2\pi_2 > 0, 3\pi_2 > 0\}$. There are 24 subcones S_{th}, $t = 1, 2$, $h = 1, \cdots, 12$. However, it can be verified that there are only 4 nonempty subcones S_{13}, S_{12}, S_{22}, and $S_{2,11}$. Therefore, the solution space S is partitioned into four convex subcones S_{13}, S_{12}, S_{22}, and $S_{2,11}$ as shown in Fig. 3. The union of the four subcones is S. Consider subcone S_{13}, $\bar{d}_1$ is the minimum dependence vector and $\bar{p}_3$ is the maximum projection vector; the objective function over S_{13} is

$$f = \frac{\Pi\bar{p}_3}{\Pi\bar{d}_1} = \frac{[\pi_1, \pi_2][-2s, 2s]^T}{[\pi_1, \pi_2][2, 2]^T} = \frac{s(-\pi_1 + \pi_2)}{\pi_1 + \pi_2}$$

$$\text{subject to} \begin{cases} \Pi\bar{d}_1 \le \Pi\bar{d}_2 \\ \Pi\bar{p}_h \ge \bar{p}_i, \qquad i = 1, \cdots, 12 \\ \Pi D > \bar{0}. \end{cases}$$

This is a linear fractional programming problem and one of the extreme directions is the optimal solution of f over S_{13} [14, p. 471]. By inspection, the set of extreme directions is $\{[0, 1], [1, 2], [1, 1]\}$ which is exactly the candidate set C^o constructed by Procedure 3.1 in Example 3.1. To illustrate Lemma 5, $S_{12} = \{\Pi: \Pi(\bar{d}_1 - \bar{d}_2) \le 0, \Pi(\bar{p}_i - \bar{p}_2) \le 0, i = 1, \cdots, 12, \Pi \in S\} = \{\Pi: \Pi(\bar{p}_3 - \bar{p}_2) \le 0, \Pi(\bar{d}_1 - \bar{d}_2) \le 0, \Pi \in S\}$. Clearly, $\Pi^e \in S_{12}$ is an extreme direction of S_{12} if and only if it satisfies one of the following equations

$$\begin{cases} \Pi(\bar{d}_1 - \bar{d}_1) = 0 \\ \Pi(\bar{p}_3 - \bar{p}_2) = \Pi[-s, 0]^T = 0 \end{cases} \tag{3.11}$$

$$\begin{cases} \Pi(\bar{d}_1 - \bar{d}_1) = 0 \\ \Pi(\bar{d}_1 - \bar{d}_2) = \Pi[2, -1]^T = 0. \end{cases} \tag{3.12}$$

Therefore, there are two extreme directions $[0, 1]$ corresponding to (3.11) and $[1, 2]$ corresponding to (3.12). To illustrate Lemma 6, consider the extreme direction $\Pi^e = [0, 1]$ determined by (3.11). In this case, $\Pi^e(\bar{d}_1 - \bar{d}_1) = 0$ corresponds to (3.8a) and $\Pi^e(\bar{p}_3 - \bar{p}_2) = 0$ corresponds to (3.8b) and $k' = 1$. The following illustrates all statements of Lemma 6 one by one. The vector $\bar{d}_1$ is a minimum dependence vector of Π^e; the norm intersection $B_2 = sp\{[0, -1]\}$ found in Example 3.1 is the one that Π^e belongs to and $k = k' = 1$; $B = [0, -1]$, $H = [0, 0]^T$, and $\text{rank}(BH) = 0 = k - 1$; and Π^e can be obtained equivalently from equations $\Pi^e = \alpha[0, -1]$ and $\alpha[0] = 0$ corresponding to (3.9) and (3.10), respectively. End of example.

Proof of Theorem 3.1: By Lemmas 1–4, the solution space S is a convex cone; S is partitioned into convex subcones S_{th}, $t = 1, \cdots, m$ and $h = 1, \cdots, q$; and over S_{th} one of the extreme directions is optimal. Therefore, one of the extreme directions of S_{th}, $t = 1, \cdots, m$ and $h = 1, \cdots, q$, must be an optimal direction of the objective function f described by (2.5) over S. By Lemmas 5 and 6, $\Pi^e \in S_{th}$ is an extreme direction of S_{th} if and only if it is a solution of (3.8), or equivalently, a solution of (3.9) and (3.10) that has only one linearly independent solution. In other words, $\Pi^e \in S_{th}$ is an extreme direction of S_{th} if and only if it is the only one linearly independent solution of (3.9) and (3.10) for some k-dimensional norm intersection of J, where $1 \le k \le \text{rank}(D)$, and some k linearly independent dependence vectors that are minimum dependence vectors of Π^e and, in (3.10), $\text{rank}(BH) = k - 1$. Now, what Procedure 3.1 does is to consider all possible combinations of norm intersections with all distinct sets of linearly independent dependence vectors, obtain a row vector Π_l by solving (3.9) and (3.10), check the feasibility of Π_l, rank of matrix BH and if the dependence vectors are minimum. If Π_l is feasible, $\text{rank}(BH) = k - 1$

nd the k dependence vectors are minimum vectors of Π_l, hen it is included in C^o. So $\Pi^e \in S_{th}$ is an extreme direction of S_{th}, $t = 1, \cdots, m$ and $h = 1, \cdots, q$, if and only if it s included in C^o which means that the optimal solution Π^o nust be in C^o. □

Complexity Analysis

The complexity of Procedure 3.1 is a constant in terms of he cardinality of the index set J, i.e., it is independent of he size of the algorithm. However, it is a function of a, m, and n, i.e., of the number of rows of A, the number of the dependencies, and the dimension of the index set, espectively. Clearly, either Step 1 or Step 5 dominates the vhole procedure. Step 1 identifies all norm intersections of J. For $1 \le k \le \min\{n-1, a\}$, there are at most $\binom{a}{k}$ distinct k-dimensional norm spaces of J. For $k = \min\{n, a\}$, here is at most one k-dimensional norm space that is $\mathbf{R}^n$ f $a \ge n$. So, there are at most $1 + \sum_{k=1}^{\min\{a,n-1\}} \binom{a}{k}$ norm paces of J and there are at most $(1 + \sum_{k=1}^{\min\{a,n-1\}} \binom{a}{k})^2$ istinct norm intersections of J. To find each norm intersection, at most $O(n^3)$ operations are needed to solve (3.3). Therefore, the total operations that Step 1 takes is $O((1 + \sum_{k=1}^{\min\{a,n-1\}} \binom{a}{k})^2 n^3)$. For Step 5, if it is assumed that there re m_k distinct k-dimensional norm intersections of J, then here are at most $\sum_{k=1}^{\text{rank}(D)} m_k \binom{m}{k}$ iterations. Each iteration eeds $O(k^3)$ operations to solve (3.10). Therefore, Step 5 akes $O(\sum_{k=1}^{\text{rank}(D)} m_k \binom{m}{k} k^3)$ operations and the complexity f Procedure 3.1 is bounded above by

$$O\left(\left(1 + \sum_{k=1}^{\min\{a,n-1\}} \binom{a}{k}\right)^2 n^3\right) + O\left(\sum_{k=1}^{\text{rank}(D)} m_k \binom{m}{k} k^3\right). \quad (3.13)$$

ue to the fact that $\sum_{k=0}^{n} \binom{n}{k} = 2^n$, it follows that $1 + \sum_{k=1}^{\min\{a,n-1\}} \binom{a}{k} \le \sum_{k=1}^{\min\{a,n\}} \binom{a}{k} < \sum_{k=0}^{a} \binom{a}{k} = 2^a$ and $\sum_{k=1}^{\text{rank}(D)} \binom{m}{k} < \sum_{k=1}^{m} \binom{m}{k} = 2^m$. Because there are at most 2^a norm spaces of J, there are at most $2^a - 1$ distinct k-imensional norm intersections of J, i.e., $m_k < 2^a$. Therefore, he complexity of Procedure 3.1 is bounded above by

$$O(2^{2a} n^3) + O(2^a n^3 2^m) \le O\left(2^a 2^{\max\{a,m\}} n^3\right). \quad (3.14)$$

When $a = 2n > m$, the complexity of Procedure 3.1 is ounded above by

$$O(2^{4n} n^3). \quad (3.14a)$$

s it can be seen from the derivation, the upper bound of rocedure 3.1 given by (3.14) is very loose. It is remarked that e procedure takes time exponential in the number of rows of e index constraint matrix A and the number of dependence ectors of the algorithm rather than in the cardinality of its dex set, i.e., it is independent of the size of the algorithm. most cases of practical interest the values of a and m are t large and, as mentioned before, when the index set of the algorithm is a hyperparallelepiped, Step 1 of Procedure 3.1 can be omitted. This reduces the complexity of Procedure 3.1 to $O(2^{2n} n^3)$ as explained later in Section V. It follows that this procedure is efficient for most algorithms.

IV. Comparison to a Linear Programming Approach

This section discusses an alternative procedure, based on linear programming, to find the time-optimal linear schedule vector Π^o and compares its complexity with that of Procedure 3.1. The comparison indicates that Procedure 3.1 is less computationally expensive than the linear programming approach. However, because there are many existing techniques to implement linear programming problems, it might be an alternative approach for some algorithms. Variations of the linear programming approach presented in this section are studied in [19], [27], and [28].

As a direct consequence of Lemmas 1, 2, and 3, Problem 2.1 can be formulated as a set of linear programs as follows. In each S_{th}, a subset of the solution space S defined in (3.7), Problem 2.1 is described as

$$\min \frac{\Pi \bar{p}_h}{\Pi \bar{d}_t}$$
$$\text{subject to} \begin{cases} \Pi \bar{d}_t \le \Pi \bar{d}_i, & i = 1, \cdots, m \\ \Pi D > \bar{0} \\ \Pi(\bar{p}_h - \bar{p}_l) \ge 0, & l = 1, \cdots, q. \end{cases} \quad (4.1)$$

As mentioned before, Π is a function of directions of the cone S_{th}. Clearly, every direction Π corresponds to a point Π' such that $disp\Pi' = \Pi' \bar{d}_t = 1$. In other words, it is good enough to consider only these feasible solutions $\Pi' \in S_{th}$ such that $disp\Pi' = \Pi' \bar{d}_t = 1$. Therefore, (4.1) is equivalent to

$$\min \frac{\Pi \bar{p}_h}{\Pi \bar{d}_t}$$
$$\text{subject to} \begin{cases} \Pi \bar{d}_t = 1 \\ \Pi D \ge \bar{1} \\ \Pi(\bar{p}_h - \bar{p}_l) \ge 0, & l = 1, \cdots, q. \end{cases} \quad (4.2)$$

The constraints $\Pi D > \bar{0}$ and $\Pi \bar{d}_t \le \Pi \bar{d}_i$, $i = 1, \cdots, m$, in (4.1) are implied by $\Pi D \ge \bar{1}$ and $\Pi \bar{d}_t = 1$. Because the denominator of the objective function is equal to one, (4.2) is equivalent to

$$\min \Pi \bar{p}_h$$
$$\text{subject to} \begin{cases} \Pi \bar{d}_t = 1 \\ \Pi D \ge \bar{1} \\ \Pi(\bar{p}_h - \bar{p}_l) \ge 0, & l = 1, \cdots, q. \end{cases} \quad (4.3)$$

Clearly, (4.3) is a linear programming problem. Let Π_{th} be the optimal solution of linear program (4.3), then the optimal solution Π^o of Problem 2.1 belongs to the set $\{\Pi_{th}, t = 1, \cdots, m, \text{ and } h = 1, \cdots, q\}$. In other words, Π^o can be found by the following procedure.

Procedure 4.1 (Finding Π^o by linear program approach):

Input: Algorithm (J, D) whose index set J is bounded.

Output: The optimal solution Π^o of Problem 2.1.

Step 1: Identify all extreme points $\bar{e}_i$, $i = 1, \cdots, e$, $e \in N^+$, of index set J.

Step 2: Construct the set of projection vectors $P = \{\bar{p}_h = \bar{e}_i - \bar{e}_k : 1 \le i, k \le e\}$. Let $|P| = q$.

Step 3: For projection vector $\bar{p}_h$ and dependence vector $\bar{d}_t$, $h = 1, \cdots, q$ and $t = 1, \cdots, m$, formulate linear program (4.3) and find the optimal solution Π_{th} of (4.3) by applying existing techniques (e.g., simplex method).

Step 4: Compare the total execution times by Π_{th}, $t = 1, \cdots, m$ and $h = 1, \cdots, q$. The one with minimum execution time is the optimal solution Π^o. Stop.

There exist many techniques to solve linear programs (4.3) such as the simplex method and the ellipsoid method [29]. Consider a linear program $\max\{C'\bar{x} : A'\bar{x} \le \bar{b}'\}$ and its dual $\min\{Y\bar{b}' : YA' = C', Y \ge 0\}$ where $C' \in Q^{1\times l}$, $A' \in Q^{n\times l}$ and $\bar{b}' \in Q^{n\times 1}$ are given and $Y \in Q^{1\times n}, \bar{x} \in Q^{l\times 1}$ are unknown vectors. The complexity of the ellipsoid method for this linear program is $O(l^8 \cdot \log^2 T)$ where T is the maximum absolute value of the entries in A' or $\bar{b}'$ and l is the number of columns of matrix A' [29, p. 170]. The complexity of the simplex method for the worst case of this linear program is $O(2^l n^3)$ [29, p. 139].

The complexity of the linear programming approach is as follows. As defined in Assumption 2.1, integer a is the number of rows of the index constraint matrix A. Clearly, there are at most $\binom{a}{n}$ extreme points of the index set J. Therefore, to find all these extreme points, at most $O\left(\binom{a}{n} n^3\right)$ operations are needed. There are at most $\left(\binom{a}{n} - 1\right)^2$ projection vectors (notice that $\bar{e}_i - \bar{e}_k$ is different from $\bar{e}_k - \bar{e}_i$ in general), i.e., $q \le \left(\binom{a}{n} - 1\right)^2$. So, for Step 2 to construct the set of projection vectors, at most $O\left(\left(\binom{a}{n} - 1\right)^2 \cdot n\right)$ operations are needed. In the worst case, there are $\left(\binom{a}{n} - 1\right)^2 m$ linear programs each of which is described by (4.3). Linear program (4.3) can be rewritten as

$$\min\{\Pi \bar{p}_h : \Pi \bar{d}_t = 1, \Pi A'' \ge C''\} \tag{4.4}$$

where $A'' = [D, \bar{p}_h - \bar{p}_1, \cdots, \bar{p}_h - \bar{p}_q]$ and $C'' = [\bar{1}, \bar{0}]$. There are $l = m + q$ columns in matrix A''. Linear program (4.4) can be transformed into standard linear program $\min\{Y\bar{b}' : Y'A' = C', Y \ge 0\}$ by some techniques (e.g., slack variables or surplus variables [31, p. 12]) and there at least $l = m + q$ columns in A'.

If the simplex method is applied to linear program (4.4), then the complexity of solving one linear program in Step 3 is $O(2^l n^3)$ where $l = m + q = m + \left(\binom{a}{n} - 1\right)^2$ is the number of columns of A''. Therefore, the number of operations needed by Step 3 is $O(m \cdot q \cdot 2^{m+q} \cdot n^3) = O(m(\binom{a}{n}-1)^2 2^{m+(\binom{a}{n}-1)^2} n^3)$ where $m \cdot q$ is the number of linear programs. Clearly, Step 3 dominates the whole procedure and the complexity of Procedure 4.1 by the simplex method is

$$O\left(m\left(\binom{a}{n} - 1\right)^2 2^{m+\left(\binom{a}{n}-1\right)^2} n^3\right). \tag{4.5}$$

If $a = 2n$, then $\binom{a}{n}^2 = \left(\sum_{k=0}^{n} \binom{n}{k}^2\right)^2 \gg \sum_{k=0}^{n} \binom{n}{k} = 2^n$ and, when the simplex method is used, the upper bound for the complexity of Procedure 4.1 is at least

$$O\left(2^n 2^m 2^{2^n} m n^3\right). \tag{4.6}$$

If the ellipsoid method is applied to solve linear program (4.4), the complexity of the linear programming approach is $O(l^8 \log^2 T \cdot m \cdot q)$ where $l = m + q = m + (\binom{a}{n} - 1)^2$ is the number of columns of A'' in (4.4) and $m \cdot q$ is the number of linear programs. As mentioned before, T is the maximum absolute value of the entries in A'' or $\bar{b}'$. Because some entries of extreme points are linear functions of entries of the size vector $\bar{b}$ (e.g., extreme point $\bar{e}_4$ in Fig. 1), some entries of the projection vectors and matrix A'' are linear functions of entries of the size vector $\bar{b}$ which characterizes the cardinality of the index set (size of the algorithm). Therefore, in addition to a large proportionality constant, the complexity of the ellipsoid method is a logarithmic function of the cardinality of the index set J which is not desirable.

Compared to Procedure 4.1, Procedure 3.1 has at least two advantages. First, Procedure 3.1 is computationally less expensive than Procedure 4.1. This is clearly shown by (3.14) and (4.5) and (3.14a) and (4.6). One of the reasons for this is that the linear programming approach, possibly, processes some extreme directions of S_{th} more than once if these extreme directions belong to more than one subcone S_{th}. This can be illustrated by Fig. 3 that shows the solution space of the algorithm of Example 2.1. In Fig. 3, extreme direction $[0, 1]$ belongs to both S_{12} and S_{13}. So, using the linear programming approach, that direction is considered twice, one by the linear program for S_{12} and the other by the linear program for S_{13}. In contrast, it is processed only once by Procedure 3.1.

Second, as mentioned in Section III-A, Procedure 3.1 can be implemented at compile-time because it does not require knowledge of the size vector $\bar{b}$. It is necessary to apply Procedure 3.1 only once to get all solutions for all possible different instances of the algorithm. In contrast, for the linear programming approach, it needs to know the size vector $\bar{b}$ to determine extreme points and projection vectors to apply the simplex method. So, the linear programming approach can be applied only when the size vector $\bar{b}$ is known, possibly, at run-time. For each instance of the same algorithm, in general, Procedure 4.1 may have to be applied once to find the optimal solution.

V. Solution of the Time Optimal Linear Schedule Problem—the Hyperparallelepiped Index Set Case

In most applications the index set J is simply a hyperparallelepiped (see Definition 5.1 below). Because of the regularity of the index set, it is reasonable to expect Procedure 3.1 to become easier for this class of algorithms. In [12], a solution of Problem 2.1 is proposed for algorithms where the index set J is a hyperparallelepiped. In this section, some of the results in [12] are restated as Corollary 5.1 and proven in a different way. Their notations are used as much as possible in this section and introduced next.

Throughout this section the shape of the index set of all algorithms under consideration is assumed to be like a hyperparallelepiped, i.e., defined as follows.

Definition 5.1 (Hyperparallelepiped index set): An index set J is a hyperparallelepiped if

$$J = \{(j_1, \cdots, j_n)^T : 0 \le j_i \le s_i, j_i \in Z, s_i \in N^+, i = 1, \cdots, n\}. \quad (5.1)$$

Clearly, if J is a hyperparallelepiped, then $J = \{\bar{j} : A\bar{j} \le \bar{b}, \bar{j} \in Z^n\}$ where $A = \begin{bmatrix} I \\ -I \end{bmatrix}$, $\bar{b} = \begin{bmatrix} \bar{s} \\ \bar{0} \end{bmatrix}$, and $\bar{s} = [s_1, \cdots, s_n]^T$. A norm vector for any boundary surface of J is of the form E_i or $-E_i$, $1 \le i \le n$, where, as defined in Section II, E_i denotes the row vector whose entries are all zeros except that the ith entry is one. Any norm space has the form $sp\{E_{r_1}, \cdots, E_{r_l}\}$ where $1 \le r_1, \cdots, r_l \le n$ and $1 \le l \le n$. Therefore, any norm intersection can choose $\{E_{r_1}, \cdots, E_{r_k}\}$, $1 \le k \le n$, as its basis and is equal to some norm space.

Let $D(c_1 \cdots c_x / r_1 \cdots r_y)$ denote the submatrix of D containing the elements in columns $c_1, \cdots, c_x$ and rows $r_1, \cdots, r_y$, i.e., it contains the elements of D at the intersections of columns $c_1, \cdots, c_x$ and rows $r_1, \cdots, r_y$. If $D(c_1 \cdots c_k / r_1 \cdots r_k)$ is nonsingular, an integer row vector $V = [v_1, \cdots, v_k] \in Z^{1 \times k}$ is defined as $V = \beta \bar{1} D^{-1}(c_1 \cdots c_k / r_1 \cdots r_k)$ where β is a positive integer such that $gcd(v_1, \cdots, v_k) = 1$. In other words, V is a vector whose entries are the sums of the corresponding columns of $D^{-1}(c_1 \cdots c_k / r_1 \cdots r_k)$ scaled so that they are integers with the greatest common divisor equal to the unity. If $D(c_1 \cdots c_k / r_1 \cdots r_k)$ is nonsingular, then define $\Pi(c_1 \cdots c_k / r_1 \cdots r_k) = VB$, where $B = \begin{bmatrix} E_{r_1} \\ \cdots \\ E_{r_k} \end{bmatrix}$. In words, the subvector $[\pi_{r_1}, \cdots, \pi_{r_k}]$ of $\Pi(c_1 \cdots c_k / r_1 \cdots r_k)$ is the same as V and the remaining entries of $\Pi(c_1 \cdots c_k / r_1 \cdots r_k)$ are zero. Finally, C_k^o denotes the set $\{\Pi(c_1 \cdots c_k / r_1 \cdots r_k) : 1 \le k \le n, 1 \le c_1 < \cdots < c_k \le m, 1 \le r_1 < \cdots < r_k \le n\}$. The following example illustrates the notations and concepts just introduced.

Example 5.1: Consider the algorithm (J, D) of Example 3.4 where $J = \{[j_1, j_2]^T : 0 \le j_1 \le s_1, 0 \le j_2 \le s_2, s_1, s_2, j_1, j_2 \in Z\}$ and $D = \begin{bmatrix} 2 & 0 \\ 2 & 3 \end{bmatrix}$. This algorithm has the same dependence matrix as the algorithm of Example 2.1. According to the definitions just introduced, $D(1/1) = [2]$ (contains the entry in the first column and the first row), its corresponding $V = \beta \bar{1} D^{-1}(1/1) = [1]$ where $\beta = 2$, $B = E_1 = [1, 0]$, and $\Pi(1/1) = VB = [1, 0]$; $D(1/2) = [2]$, its corresponding $V = \beta \bar{1} D^{-1}(1/2) = [1]$ where $\beta = 2$, $B = E_2 = [0, 1]$ and $\Pi(1/2) = VB = [0, 1]$; and $D(12/12) = D$, its corresponding $V = \beta \bar{1} D^{-1} = [1, 2]$ where $\beta = 6$, $B = I$, and $\Pi(12/12) = VB = V = [1, 2]$. End of example.

Corollary 5.1: If the index set J of algorithm (J, D) is a hyperparallelepiped defined by (5.1), then the optimal solution Π^o of Problem 2.1 for (J, D) belongs to C_h^o, i.e.,

$$\Pi^o \in C_h^o = \{\Pi(c_1 \cdots c_k / r_1 \cdots r_k) : 1 \le k \le n, 1 \le c_1 < \cdots < c_k \le m, 1 \le r_1 < \cdots < r_k \le n\}. \quad (5.2)$$

Proof: Provided in Appendix.

Procedure 5.1 (Construction of C_h^o):

Input: Algorithm (J, D) where J is a hyperparallelepiped.
Output: A finite candidate set C_h^o containing the optimal solution of Problem 2.1.
Step 1: $k = 1$, $l = 1$.
Step 2: $C_k^h = \emptyset$.
Step 3: Pick an unprocessed combination of k elements from $\{1, \cdots, n\}$. Denote it $\{r_1, \cdots, r_k\}$.
Step 4: Pick an unprocessed combination of k elements from $\{1, \cdots, m\}$. Denote it $\{c_1, \cdots, c_k\}$.
Step 5: If $D(c_1 \cdots c_k / r_1 \cdots r_k)$ is not singular, then $\Pi_l = \Pi(c_1 \cdots c_k / r_1 \cdots r_k)$ and $C_k^h = C_k^h \cup \{\Pi_l\}$, $l = l + 1$.
Step 6: Check if all distinct combinations of k elements from $\{1, \cdots, m\}$ have been processed. If not, go to Step 4.
Step 7: Check if all distinct combinations of k elements from $\{1, \cdots, n\}$ have been processed. If not, go to Step 3.
Step 8: If $k < \text{rank}(D)$, then $k = k + 1$, go to Step 2.
Step 9: $C_h^o = \bigcup_{k=1}^{\text{rank}(D)} C_k^h$. Stop.

Due to the fact that all norm intersections of J are known, Procedure 5.1 does not compute distinct norm intersections. This makes its complexity lower than Procedure 3.1. Clearly, Step 5 dominates the whole procedure. For Step 5, there are $\binom{n}{k}$ distinct combinations of k elements from $\{1, \cdots, n\}$ and there are $\binom{m}{k}$ distinct combinations of k elements from $\{1, \cdots, m\}$. So there are $\sum_{k=1}^{\text{rank}(D)} \binom{n}{k}\binom{m}{k} = \binom{m+n}{\text{rank}(D)} - 1$ iterations for Step 5. Each iteration needs $O(k^3)$ operations to compute the inverse $D^{-1}(c_1 \cdots c_k / r_1 \cdots r_k)$ if it is nonsingular. Therefore, the complexity for Procedure 5.1 is bounded above by $O(((\binom{m+n}{\text{rank}(D)}) - 1)n^3)$. If $\text{rank}(D) = n = m$, then $\sum_{k=1}^{\text{rank}(D)} \binom{n}{k}\binom{m}{k} = \sum_{k=1}^{n} \binom{n}{k}^2 \le (\sum_{k=1}^{n} \binom{n}{k})^2 = (2^n)^2 = 2^{2n}$ and the complexity of Procedure 5.1 is bounded above by $O(2^{2n} n^3)$.

Some comments are made here regarding how the candidate set C_h^o constructed by Procedure 5.1 is related to C^o constructed by Procedure 3.1. First, each candidate $\Pi(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$ in C_h^o corresponds to the combination of the norm intersection $sp\{E_{r_1}, \cdots, E_{r_k}\}$ and the set of dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$. This is shown in the proof of Corollary 5.1 provided in the Appendix. So $\Pi(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$ belongs to the solution space of (3.4). This can be verified as follows. Let $\Pi = \Pi(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$, then $\Pi = VB$ where $V = \beta \bar{1} D^{-1}(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$ and $B = \begin{bmatrix} E_{r_1} \\ \cdots \\ E_{r_k} \end{bmatrix}$. Clearly, $B[\bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}, \bar{d}_t] = D(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$. So $\Pi[\bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}, \bar{d}_t] = \beta \bar{1}$, i.e., $\Pi \bar{d}_{t_1} = \cdots = \Pi \bar{d}_{t_{k-1}} = \Pi \bar{d}_t = \beta$. Therefore, Π is a solution of (3.4).

Second, the candidate set C^o constructed by Procedure 3.1 is a subset of C_h^o, i.e., $C^o \subseteq C_h^o$ for algorithms whose index sets are hyperparallelepiped. Let $\Pi \in C^o$ be determined by a k-dimensional norm intersection B and k linearly independent dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$. As

mentioned before, a basis for B is $\{E_{r_1}, \cdots, E_{r_k}\}$. Then $\Pi = \Pi(t_1 \cdots t_{k-1}t/r_1 \cdots r_k) \in C_h^o$. As a matter of fact, $C^o = \{\Pi: \Pi \in C_h^o, \Pi \text{ or } -\Pi \text{ is feasible and } \bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}} \text{ are minimum dependence vectors of } \Pi\}$. This is true because each $\Pi(t_1 \cdots t_{k-1}t/r_1 \cdots r_k)$ corresponds to a combination of a k-dimensional norm space $sp\{E_{r_1}, \cdots, E_{r_k}\}$ with k linearly independent dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$ and vice versa and C_h^o includes all the row vectors corresponding all combinations as long as $D^{-1}(t_1 \cdots t_{k-1}t/r_1 \cdots r_k)$ is nonsingular, i.e., the condition $\text{rank}(BH) = k - 1$ in Section III-A is met.

Third, for $\Pi(t_1 \cdots t_{k-1}t/r_1 \cdots r_k)$, the order by which $t_1, \cdots, t_{k-1}$ and t appear and the order by which $r_1, \cdots, r_k$ appear do not matter. As an example, $\Pi(t_1 \cdots t_{k-1}t/r_1 \cdots r_k) = \Pi(t_2t_1 \cdots t_{k-1}t/r_1 \cdots r_k) = \Pi(t_2t_1 \cdots t_{k-1}t/r_2r_1 \cdots r_k) = \Pi(t_1 \cdots t_{k-1}t/r_2r_1 \cdots r_k)$. This is due to the fact that they are all determined by the same combination of a k-dimensional norm intersection $sp\{E_{r_1}, \cdots, E_{r_k}\}$ with a set of k linearly dependence vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$. Next, an example is used to illustrate Procedure 5.1.

Example 5.2: Consider the algorithm of Example 5.1. Let $k = 1$, there are two possible combinations of one element from $\{1, \cdots, n\}$ and two possible combinations of one element from $\{1, \cdots, m\}$ $(n = m = 2)$. So, $D(1/1)$, $D(1/2)$, $D(2/1)$, and $D(2/2)$ are processed. Each corresponding vector is obtained as follows. $D(1/1) = [2]$ and $\Pi(1/1) = [1, 0]$; $D(2/1) = [0]$ is ignored because it is singular; $D(1/2) = [2]$ and $\Pi(1/2) = [0, 1]$; $D(2/2) = [3]$ and $\Pi(2/2) = [0, 1]$. For $k = 2$, there is only one combination, i.e., $D(12/12) = D$ and $\Pi(12/12) = [1, 2]$. So $C_h^o = \{[1, 0], [0, 1], [1, 2]\}$. $\Pi(1/1)$ is not feasible. If it is removed from C_h^o, then C^o is obtained as $\{[0, 1], [1, 2]\}$ which is the same as the one constructed by Procedure 3.1 in Example 3.4. The total execution time by each candidate is as follows.

$$t([0, 1]) = \left\lceil \frac{[0, 1]([0, s_2]^T - [0, 0]^T) + 1}{2} \right\rceil = \left\lceil \frac{s_2 + 1}{2} \right\rceil$$

$$t([1, 2]) = \left\lceil \frac{[1, 2]([s_1, s_2]^T - [0, 0]^T) + 1}{6} \right\rceil = \left\lceil \frac{s_1 + 2s_2 + 1}{6} \right\rceil.$$

If $s_1 \gg s_2$, then the optimal solution of Problem 2.1 is $\Pi^o = \Pi(1/2) = [0, 1]$. Π^o is perpendicular to the surface $\text{surf}\{[0, 1]\}$. Compared to Example 3.1, these two algorithms have different optimal linear schedule vectors although they have the same dependence matrix. This is due to the fact that they have different index sets. End of example.

VI. Conclusions and Future Work

In summary, this paper provides a solution to the problem of identifying optimal linear schedules for the execution of algorithms with uniform dependences. The complexity of the proposed procedure is independent of the size of the algorithm and is exponential in the dimension of its index set. For practical algorithms the index set has rather small dimension and the procedure is therefore very efficient. In comparison to extant work, this contribution considers a larger class of index sets and linear schedules. It is also shown that, when index sets are hyperparallelepipeds, the procedure is considerably simpler and faster. The proposed optimization procedure can also be used to identify optimal schedules for the execution of a specific computation [16].

In MIMD machines, the total execution time of an algorithm consists not only of the time required to execute all of its computations but also of the time spent in coordinating concurrent operations, e.g., in data communication and synchronization. The partitioning techniques studied in [15], [25], and [26] are potentially applicable to the problems of selecting the linear schedules which minimize the time spent in the coordination of concurrent operations. Together with the scheduling approach discussed in this paper, those techniques may be of use to identify optimal schedules for uniform dependence algorithms executed in MIMD machines. This topic will be addressed in future research.

Appendix

Proof of Lemma 1: By definition,

$$S = \bigcap_{i=1}^{m} \{\Pi: \Pi\bar{d}_i > 0\}. \tag{a.1}$$

Clearly, $\{\Pi: \Pi\bar{d}_i > 0\}$ is a convex cone. By Theorem 2.5 in [17] which states that the intersection of an arbitrary collection of convex cones is a convex cone, S is a convex cone. For the objective function f in (2.5), $f(\Pi) = f(\lambda\Pi)$ for any nonzero constant λ. Therefore, f is a function of directions in S. □

Proof of Lemma 2: Obviously, S_{th} is a convex cone because it is an intersection of convex cones. By definition of S_{th}, for all directions $\Pi \in S_{th}$, $\Pi\bar{d}_t \leq \Pi\bar{d}_i$, $i = 1, \cdots, m$ and $\Pi\bar{p}_h \geq \Pi\bar{p}_k$, $k = 1, \cdots, q$. Therefore, $\bar{d}_t$ is the minimum dependency of $\Pi \in S_{th}$, and $\bar{p}_h$ is the maximum projection vector of $\Pi \in S_{th}$, i.e., $\Pi\bar{d}_t = \min\{\Pi\bar{d}_i : \bar{d}_i \in D\}$ and $\Pi\bar{p}_h = \max\{\Pi\bar{p}_i : \bar{p}_i \in P\} = \max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}$. Thus,

$$f = \frac{\max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}}{\min\{\Pi\bar{d}_i : \bar{d}_i \in D\}} = \frac{\Pi\bar{p}_h}{\Pi\bar{d}_t} \qquad \Pi \in S_{th}.$$

□

Proof of Lemma 3: For every $\Pi \in S$, there must exist $\bar{d}_t$ and $\bar{p}_h$ such that $\bar{d}_t$ is the minimum dependency and $\bar{p}_h$ is the maximum projection vector which implies that there exists a subcone S_{th} such that $\Pi \in S_{th}$. So $S \subseteq \bigcup_{t=1}^{m} \bigcup_{h=1}^{q} S_{th}$. By definition of S_{th} [see (3.6)], for every $\Pi \in S_{th}$, $\Pi \in S$. Therefore, $S_{th} \subseteq S$ which means that $S \supseteq \bigcup_{t=1}^{m} \bigcup_{h=1}^{q} S_{th}$. Therefore, $S = \bigcup_{t=1}^{m} \bigcup_{h=1}^{q} S_{th}$. □

Proof of Lemma 4: In S_{th}, Problem 2.1 can be formulated as

$$\min f = \frac{\Pi\bar{p}_h}{\Pi\bar{d}_t} \qquad \Pi \in S_{th}$$

where S_{th} is defined by (3.7). According to [11, Section 11.4], this is a linear fractional programming problem and one of the extreme directions is the optimal direction of f over S_{th}. □

Proof of Lemma 5: The idea behind this proof is as follows. First, if Π^e is an extreme direction and there does not exist a set of $n-1$ linearly independent vectors $\bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}, \bar{p}_{h_1} - \bar{p}_h, \cdots, \bar{p}_{h_{n-k'}} - \bar{p}_h$ such that Π^e satisfies (3.8) then Π^e can be written as a positive linear combination of two distinct directions $\Pi_1, \Pi_2 \in S_{th}$ (Π_1 and Π_2 are distinct if $\Pi_1 \neq \alpha\Pi_2$ for any nonzero constant α). By definition of an extreme direction [14, p. 55], Π^e cannot be an extreme direction which is a contradiction. Second, it is shown that if there exists a set of $n-1$ linearly independent vectors $\bar{l}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}, \bar{p}_{h_1} - \bar{p}_h, \cdots, \bar{p}_{h_{n-k'}} - \bar{p}_h$ such that Π^e satisfies (3.8), then Π^e cannot be written as any positive linear combination of two distinct directions of S_{th}, i.e., Π^e is an extreme direction. The formal proof is as follows.

1) ($\Rightarrow$) A direction Π of a convex set Ω is called an extreme direction if it cannot be written as a positive linear combination of two distinct directions in Ω, that is, if $\Pi = \lambda_1\Pi_1 + \lambda_2\Pi_2$ where $\lambda_1, \lambda_2 > 0$ and $\Pi_1, \Pi_2 \in \Omega$, then $\Pi_1 = \alpha\Pi_2$ for some constant $\alpha > 0$ [14, p. 55]. By definition, $S_{th} = \{\Pi : \Pi[\bar{d}_t - \bar{d}_1, \cdots, \bar{d}_t - \bar{d}_m, \bar{p}_1 - \bar{p}_h, \cdots, \bar{p}_q - \bar{p}_h] \leq \bar{0}, -\Pi[\bar{d}_1, \cdots, \bar{d}_m] < \bar{0}\}$. Without loss of generality, let $\bar{c}_i = \bar{l}_t - \bar{d}_i$ for $i = 1, \cdots, m$, and $\bar{c}_{l+m} = \bar{p}_l - \bar{p}_h$ for $l = 1, \cdots, q$, i.e., $S_{th} = \{\Pi : \Pi[\bar{c}_1, \cdots, \bar{c}_r] \leq 0, -\Pi[\bar{d}_1, \cdots, \bar{d}_m] < 0\}$, where $r = m + q$. Suppose that Π^e is an extreme direction in S_{th} and there does not exist a set of $n-1$ linearly independent vectors $\bar{c}_{t_1}, \cdots, \bar{c}_{t_{n-1}}$ such that Π^e satisfies (3.8). Because $\Pi^e \in S_{th}$, it satisfies the following equations:

$$\Pi[\bar{c}_1, \cdots, \bar{c}_r] \leq \bar{0}$$
$$-\Pi[\bar{d}_1, \cdots, \bar{d}_m] < \bar{0}.$$

Assume that there are s vectors $\bar{c}_{t_1}, \cdots, \bar{c}_{t_s}$, $s \in Z$, such that $\Pi^e\bar{c}_{t_i} = 0$, $i = 1, \cdots, s$, and $\text{rank}([\bar{c}_{t_1}, \cdots, \bar{c}_{t_s}]) = s' < n-1$. Without loss of generality, let us assume $t_i = i$, $i = 1, \cdots, s$, i.e., Π^e is a solution of (a.2) and satisfies (a.3) below:

$$\begin{cases} \Pi\bar{c}_1 = 0 \\ \cdots \\ \Pi\bar{c}_s = 0 \end{cases} \tag{a.2}$$

and

$$\begin{cases} \Pi\bar{c}_{s+1} < 0 \\ \cdots \\ \Pi\bar{c}_r < 0 \\ -\Pi\bar{d}_1 < 0 \\ \cdots \\ -\Pi\bar{d}_m < 0. \end{cases} \tag{a.3}$$

There are at least two independent solutions of (a.2) because $s' < n-1$. Next, it is shown that there exist at least two linearly independent solutions Π^e and Π_2 of (a.2) that are in S_{th}. Let Π' be a solution of (a.2) and $\Pi' \neq \gamma\Pi^e$ for any constant γ, i.e., Π' and Π^e are linearly independent. Let $\Pi_2 = \alpha'\Pi' + \alpha\Pi^e$ where $\alpha' > 0$ and $\alpha > 0$ are constants. Obviously, Π_2 satisfies (a.2). Now,

$$\Pi_2[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, \bar{d}_m] = \alpha'\Pi'[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, \bar{d}_m] + \alpha\Pi^e[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m].$$

It is always possible to choose a large enough constant α and a small enough constant α' such that $\Pi_2[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m] < \bar{0}$ because $\Pi^e[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m] < \bar{0}$. Thus, $\Pi_2 \in S_{th}$, i.e., there exist at least two linearly independent solutions of (a.2) Π_2 and Π^e that are in S_{th}. Let Π_1 be constructed as follows:

$$\Pi_1 = \lambda\Pi^e - \lambda_2\Pi_2 \quad \text{where } \lambda > 0 \quad \text{and} \quad \lambda_2 > 0 \text{ are constants.} \tag{a.4}$$

The fact that $\Pi^e \neq \gamma\Pi_2$ for any constant γ implies $\Pi_1 \neq \gamma_2\Pi_2$ for any constant γ_2. Thus, Π_1 and Π_2 are distinct. If it can be shown that $\Pi_1 \in S_{th}$, then by the definition of extreme directions, Π^e is not an extreme direction which is contrary to the assumption. Now, Π_1 is a solution of (a.2) because it is a linear combination of two solutions of (a.2). For (a.3)

$$\Pi_1[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m] = \lambda\Pi^e[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m] - \lambda_2\Pi_2[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m].$$

Again, it is always possible to choose a large enough constant λ and a small enough constant λ_2 such that $\Pi_1[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, -\bar{d}_m] < \bar{0}$ because $\Pi^e[\bar{c}_{s+1}, \cdots, \bar{c}_r, -\bar{d}_1, \cdots, \bar{d}_m] < \bar{0}$. Thus, $\Pi_1 \in S_{th}$ and Π^e can be expressed as a linear positive combination of two other distinct directions in S_{th}, i.e., $\Pi^e = \alpha_1\Pi_1 + \alpha_2\Pi_2$ where $\alpha_1 = (1/\lambda) > 0$ and $\alpha_2 = (\lambda_2/\lambda) > 0$, which means Π^e is not an extreme direction. By assumption, Π^e is an extreme direction. Therefore, there must exist a set of $n-1$ linearly independent vectors $\bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}, \bar{p}_{h_1} - \bar{p}_h, \cdots, \bar{p}_{n-k'} - \bar{p}_h$ such that Π^e satisfies (3.8).

2) ($\Leftarrow$) Conversely, suppose that there exists a set of $n-1$ linearly independent vectors $\bar{c}_1, \cdots, \bar{c}_{n-1}$ such that Π^e is a solution of (3.8) and it is not an extreme direction of S_{th}. Then Π^e can be expressed as a positive linear combination of two distinct directions of S_{th}, i.e., there exist $\Pi_1, \Pi_2 \in S_{th}$ such that $\Pi^e = \lambda_1\Pi_1 + \lambda_2\Pi_2$ where $\lambda_1, \lambda_2 > 0$ and $\Pi_1 \neq \alpha\Pi_2$ for any constant α. By bringing Π^e into (3.8), then

$$\lambda_1\Pi_1\bar{c}_i + \lambda_2\Pi_2\bar{c}_i = 0 \qquad i = 1, \cdots, n-1. \tag{a.5}$$

Because $\Pi_1, \Pi_2 \in S_{th}$, $\lambda_1\Pi_1\bar{c}_i \leq 0$ and $\lambda_2\Pi_2\bar{c}_i \leq 0$. Therefore, the only choices for Π_1 and Π_2 that satisfy (a.5) must be such that

$$\lambda_1\Pi_1\bar{c}_i = \lambda_2\Pi_2\bar{c}_i = 0 \qquad i = 1, \cdots, n-1.$$

So, Π_1, and Π_2 are solutions of (3.8). Because $\text{rank}([\bar{c}_1, \cdots, \bar{c}_{n-1}]) = n-1$, there is only one linearly independent solution of (3.8). So Π^e, Π_1, and Π_2 are linearly dependent, i.e., $\Pi^e = \alpha_1\Pi_1 = \alpha_2\Pi_2$ for some nonzero constants α_1 and α_2. This means that Π_1 and Π_2 are not distinct in contradiction with the initial assumption. Therefore, Π^e is an extreme direction. □

Proof of Lemma 6:

Proof of Statement a): Suppose $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k'-1}}$ are linearly dependent. Then $\text{rank}([\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k'-1}}]) = k' - 1$ because $\bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - d_{t_{k'-1}}$ are linearly independent.

Consider the following equation:

$$\begin{cases} \Pi\bar{d}_t = 0 \\ \Pi\bar{d}_{t_1} = 0 \\ \cdots \\ \Pi\bar{d}_{t_{k'-1}} = 0. \end{cases} \quad \text{(a.6)}$$

Let Ω_1 and Ω_2 be the solution space of (a.6) and (3.8a), respectively. If Π belongs to Ω_1, then Π belongs to Ω_2 because $\Pi\bar{d}_t - \Pi\bar{d}_{t_i} = 0$, $i = 1, \cdots, k'-1$, i.e., Π satisfies (3.8a). Thus, $\Omega_1 \subseteq \Omega_2$. Furthermore, $\Omega_1 = \Omega_2$ because $\dim\{\Omega_1\} = \dim\{\Omega_2\} = n - k' + 1$. Now because Π^e satisfies (3.8a), it belongs to Ω_2 which means it belongs to Ω_1 also. So Π^e is not feasible because $\Pi^e\bar{d}_{t_i} = 0$, $i = 1, \cdots, k'-1$ which is contrary to the assumption that $\Pi^e \in S_{th}$. Therefore, $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k'-1}}$ must be linearly independent. Moreover, $\bar{d}_t$ is a minimum dependence vector of Π^e because $\Pi^e \in S_{th}$. So $\bar{d}_t, \bar{d}_1, \cdots, \bar{d}_{k'-1}$ are minimum dependence vectors of Π^e due to the fact that $\Pi^e\bar{d}_t = \Pi^e\bar{d}_{t_1} = \cdots = \Pi^e\bar{d}_{t_{k'-1}}$.

Proof of Statement b): At this point, some additional notations are needed. Let $F_1 = \text{surf}\{A_{t_1}, A_{t_2}, \cdots, A_{t_{k_1}}\}$ and $F_2 = \text{surf}\{A_{h_1}, A_{h_2}, \cdots, A_{h_{k_2}}\}$ be boundary surfaces of J, $\mathcal{F}_r = \{\bar{x}_1 - \bar{x}_2 : \bar{x}_1, \bar{x}_2 \in F_r\}$, $r = 1, 2$, $N_1 = sp\{A_{t_1}, \cdots, A_{t_{k_1}}\}$, $N_2 = sp\{A_{h_1}, \cdots, A_{h_{k_2}}\}$ and $B = N_1 \cap N_2$. N_r is the norm space of J associated with F_r, $r = 1, 2$, B is a norm intersection of J and $\mathcal{F}_1$ and $\mathcal{F}_2$ are the solution spaces of (a.7a) and (a.7b) as follows, respectively.

$$\text{(a)} \begin{cases} A_{t_1}\bar{y} = 0 \\ \cdots \\ A_{t_{k_1}}\bar{y} = 0 \end{cases} \qquad \text{(b)} \begin{cases} A_{h_1}\bar{y} = 0 \\ \cdots \\ A_{h_{k_2}}\bar{y} = 0. \end{cases} \quad \text{(a.7)}$$

Consider the projection vectors $\bar{p}_h, \bar{p}_{h_1}, \cdots, \bar{p}_{h_{n-k'}}$ of (3.8b) and, without loss of generality, let $\bar{p}_{h_i} = \bar{p}_i$, $i = 1, \cdots, n-k'$ and $h > n - k'$. By definition of projection vectors, $\bar{p}_i$ is the difference of two extreme points of J, i.e., $\bar{p}_i = \bar{e}_{1i} - \bar{e}_{2i}$ for some extreme points $\bar{e}_{1i}$ and $\bar{e}_{2i}$, where $\bar{e}_{1i}$ is the head of $\bar{p}_i$ and $\bar{e}_{2i}$ is the tail of $\bar{p}_i$, $i = h, 1, \cdots, n-k'$. Let H_r, $r = 1, 2$, denote the convex hull of the extreme points $\bar{e}_{ri}$, $i = h, 1, \cdots, n-k'$. In other words, H_1 and H_2 are the convex hulls of all the heads and tails of projection vectors $\bar{p}_h, \bar{p}_1, \cdots, \bar{p}_{n-k'}$, respectively. Three facts are proved next.

First, it is shown that Π^e is perpendicular to both H_1 and H_2, i.e., $\Pi^e(\bar{x}_1 - \bar{x}_2) = 0$, $\bar{x}_1, \bar{x}_2 \in H_r$, $r = 1, 2$. Because $\Pi^e\bar{e}_{1i} = \max\{\Pi^e\bar{x} : \bar{x} \in R\}$ and $\Pi^e\bar{e}_{2i} = \min\{\Pi^e\bar{x} : \bar{x} \in R\}$, $i = h, 1, \cdots, n-k'$, $\Pi^e\bar{e}_{1h} = \Pi^e\bar{e}_{1i}$ and $\Pi^e\bar{e}_{2h} = \Pi^e\bar{e}_{2i}$, $i = 1, \cdots, n-k'$. If $\bar{x} \in H_r$, $r = 1, 2$, then it can be expressed as a convex combination of $\bar{e}_{ri}$, $i = h, 1, \cdots, n-k'$, i.e., $\bar{x} = \lambda_{rh}\bar{e}_{rh} + \sum_{i=1}^{n-k'} \lambda_{ri}\bar{e}_{ri}$ where λ_{ri}, $i = h, 1, \cdots, n-k'$, are nonnegative and $\lambda_{rh} + \sum_{i=1}^{n-k'} \lambda_{ri} = 1$ [14, p. 37]. Thus,

$$\begin{aligned} \Pi^e\bar{x} &= \lambda_{rh}\Pi^e\bar{e}_{rh} + \lambda_{r1}\Pi^e\bar{e}_{r1} + \cdots + \lambda_{r(n-k')}\Pi^e\bar{e}_{r(n-k')} \\ &= \Pi^e\bar{e}_{rh}\sum_i \lambda_{ri} = \Pi^e\bar{e}_{rh}. \end{aligned}$$

Therefore, for any two points $\bar{x}_1, \bar{x}_2 \in H_r$, $\Pi^e\bar{x}_1 = \Pi^e\bar{x}_2 = \Pi^e\bar{e}_{rh}$, i.e., $\Pi^e(\bar{x}_1 - \bar{x}_2) = 0$ which means that Π^e is perpendicular to both H_1 and H_2.

Second, it is shown that H_r is contained in some boundary surface of J, i.e., there exist two boundary surfaces F_1 and F_2 of J such that $H_r \subseteq F_r$, $r = 1, 2$. Notice that for any $\bar{x} \in R$ and any row vector Π, if $\Pi\bar{x} = \min\{\Pi\bar{x} : \bar{x} \in R\}$ or $\Pi\bar{x} = \max\{\Pi\bar{x} : \bar{x} \in R\}$, then $\bar{x}$ is on some boundary surface of J. As mentioned before, for any $\bar{x} \in H_1$, $\Pi^e\bar{x} = \Pi^e\bar{e}_{1h} = \max\{\Pi^e\bar{x} : \bar{x} \in R\}$. So H_1 is contained in some boundary surface of J. Similarly, because for all $\bar{x} \in H_2$, $\Pi^e\bar{x} = \Pi^e\bar{e}_{2h} = \min\{\Pi^e\bar{x} : \bar{x} \in R\}$, H_2 is contained in some boundary surface of J. Let F_r be the boundary surface of J with the largest dimension which contains H_r and is perpendicular to Π^e, i.e., $H_r \subseteq F_r$ and $\Pi^e(\bar{x}_1 - \bar{x}_2) = 0$, $\bar{x}_1$, $\bar{x}_2 \in F_r$, $r = 1, 2$.

Third, it is shown that $\dim\{\mathcal{F}_1 \cup \mathcal{F}_2\} \geq n - k'$. Now, $\bar{p}_h - \bar{p}_i = (\bar{e}_{1h} - \bar{e}_{2h}) - (\bar{e}_{1i} - \bar{e}_{2i}) = (\bar{e}_{1h} - \bar{e}_{1i}) - (\bar{e}_{2h} - \bar{e}_{2i})$, $i = 1, \cdots, n-k'$, i.e., $\bar{p}_h - \bar{p}_i$ is a linear combination of $\bar{e}_{1h} - \bar{e}_{1i}$ and $\bar{e}_{2h} - \bar{e}_{2i}$, $i = 1, \cdots, n-k'$. By assumption, $\dim\{\bar{p}_h - \bar{p}_i : i = 1, \cdots, n-k'\} = n - k'$, so $\dim\{\bar{e}_{1h} - \bar{e}_{1i}, \bar{e}_{2h} - \bar{e}_{2i} : i = 1, \cdots, n-k'\} \geq n - k'$. Let $\mathcal{H}_r = \{\bar{x}_1 - \bar{x}_2 : \bar{x}_1, \bar{x}_2 \in H_r\}$, $r = 1, 2$, then, $\dim\{\mathcal{H}_1 \cup \mathcal{H}_2\} = \dim\{\bar{e}_{1h} - \bar{e}_{1i}, \bar{e}_{2h} - \bar{e}_{2i} : i = 1, \cdots, n-k'\} \geq n - k'$. Because $F_r \supseteq H_r$, it follows that $\mathcal{F}_r \supseteq \mathcal{H}_r$, $r = 1, 2$. This means that $\dim\{\mathcal{F}_1 \cup \mathcal{F}_2\} \geq \dim\{\mathcal{H}_1 \cup \mathcal{H}_2\} \geq n - k'$.

So far, the following statement has been proven. There exist two boundary surfaces F_1 and F_2 of J to which Π^e is perpendicular and $\dim\{\mathcal{F}_1 \cup \mathcal{F}_2\} \geq n - k'$. Next, it is shown that Π^e belongs to B, the intersection of the norm spaces of F_1 and F_2 and $\dim\{B\} = k \leq k'$.

Consider (a.7a). By [18], its row space N_1 and the solution space $\mathcal{F}_1$ are orthogonal complements of each other, i.e., $N_1 \cap \mathcal{F}_1 = \emptyset$, $N_1 \cup \mathcal{F}_1 = \mathbf{R}^n$ and $X \cdot \bar{x} = 0$, $X \in N_1$ and $\bar{x} \in \mathcal{F}_1$ (X and $\bar{x}$ are row and column vectors, respectively). Because Π^e is perpendicular to F_1, i.e., $\Pi^e(\bar{x}_1 - \bar{x}_2) = 0$, $\bar{x}_1 - \bar{x}_2 \in \mathcal{F}_1$ (or $\bar{x}_1, \bar{x}_2 \in F_1$), it must be in N_1, the norm space of F_1. Vector Π^e is perpendicular to both F_1 and F_2 which means that it belongs to both N_1 and N_2. Therefore, Π^e must belong to $B = N_1 \cap N_2$, i.e., $\Pi^e = \sum_{i=1}^{k} \alpha_i B_i$ where $\{B_1, \cdots, B_k\}$ is a basis for B and α_i, $i = 1, \cdots, k$, are constants.

Because $\mathcal{F}_r$ is the orthogonal complement of N_r, the orthogonal complement of $N_1 \cap N_2$ is $\mathcal{F}_1 \cup \mathcal{F}_2$ [18] which means that

$$k = \dim\{N_1 \cap N_2\} = n - \dim\{\mathcal{F}_1 \cup \mathcal{F}_2\}.$$

Because $\dim\{\mathcal{F}_1 \cup \mathcal{F}_2\} \geq n - k'$, $k \leq n - (n - k') = k'$. So $k \leq k'$.

Now, it has been proven that there exists a k-dimensional norm space B such that (3.9) holds and $k \leq k'$. To facilitate the understanding of the proof of statement b) of Lemma 6, the most difficult one in this paper, an example is provided next.

Example a.1: Consider the algorithm of Example 2.1. As mentioned in Example 3.5, one extreme direction is identified as $\Pi^e = [0, 1]$ which satisfies the following equations:

$$\text{(a)} \quad \Pi(\bar{d}_1 - \bar{d}_1) = 0 \qquad \text{(b)} \quad \Pi(\bar{p}_3 - \bar{p}_2) = 0 \quad \text{(a.8)}$$

Equation (a.8b) corresponds to (3.8b). As indicated in Fig. 1, $\bar{p}_3 = \bar{e}_1 - \bar{e}_4$ and $\bar{p}_2 = \bar{e}_1 - \bar{e}_3$. The sets of heads and tails

are $\{\bar{e}_1\}$ and $\{\bar{e}_4, \bar{e}_3\}$, respectively. In this case, $H_1 = \{\bar{e}_1\}$ and H_2 is the line segment from $\bar{e}_3$ to $\bar{e}_4$. Clearly, Π^e is perpendicular to both H_1 and H_2; H_1 is contained in the zero-dimensional boundary surface $F_4 = \text{surf}\{A_1, A_3\}$ as shown in Fig. 1 and H_2 is contained in the one-dimensional boundary surface $F_2 = \text{surf}\{A_2\}$ as shown in Fig. 1; Π^e is perpendicular to both F_4 and F_2; $\dim\{\mathcal{F}_1 \cup \mathcal{F}_2\} = 1 = n - k'$ $n = 2, k' = 1$); Π^e belongs to the norm intersection $B = sp\{A_1, A_3\} \cap sp\{A_2\} = \mathbf{R}^2 \cap sp\{A_2\} = sp\{A_2\}$ and $\dim\{B\} = k = k' = 1$. End of example.

Proof of Statement c): Let $B = \begin{bmatrix} B_1 \\ \cdots \\ B_k \end{bmatrix}$, $H' = [\bar{d}_t - \bar{d}_t, \bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}]$ and $\bar{\alpha}^T = [\alpha_1, \cdots, \alpha_k]$. If Π in (3.8a) is substituted by $\bar{\alpha}^T B$, then $\bar{\alpha}$ can be found by solving equation

$$\bar{\alpha}^T B H' = \bar{0}. \tag{a.9}$$

$\Pi^e \neq \bar{0}$ is the only linearly independent solution satisfying (3.8). Equation (3.8) is equivalent to (3.9) and (a.9), i.e., the vector satisfying (3.9) and (a.9) must satisfy (3.8) and vice versa. If $\text{rank}(BH') > k - 1$, then $\bar{\alpha} = \bar{0}$ which means $\Pi^e = \bar{0}$. If $\text{rank}(BH') < k - 1$, then there are at least two linearly independent solutions $\bar{\alpha}_1$ and $\bar{\alpha}_2$ of (a.9). Then it can be shown that there are at least two linearly independent solutions $\bar{\alpha}_1^T B$ and $\bar{\alpha}_2^T B$ of (3.9) and (a.9), or equivalently (3.8) [30]. So far, it has been proven that if $\text{rank}(BH') > k - 1$, then $\Pi^e = \bar{0}$ and if $\text{rank}(BH') < k - 1$, then there are at least two linearly independent solutions of (3.8). In both cases, the original assumption is contradicted. So $\text{rank}(BH')$ must be $k - 1$.

Proof of Statement d): Because $\text{rank}(BH') = k - 1$, there are only $k - 1$ linearly independent column vectors in $B[\bar{d}_t - \bar{d}_{t_1}, \cdots, \bar{d}_t - \bar{d}_{t_{k'-1}}]$. Without loss of generality, let us assume that the first $k - 1$ columns are linearly independent. So, only k linearly independent vectors $\bar{d}_t, \bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}$ are needed to determine $\bar{\alpha}$. In other words, (a.9) is equivalent to (3.10). So Π^e can be obtained equivalently by (3.9) and (3.10). □

Proof of Corollary 5.1: Clearly, if J is defined by (5.1), then any norm vector has the form E_i or $-E_i$, $i \in \{1, \cdots, n\}$. Therefore, $\{E_{r_1}, \cdots, E_{r_k}\}$, where $r_1, \cdots, r_k \in \{1, \cdots, n\}$, can be chosen as the basis of any k-dimensional norm intersection of J. Let Π^e be an extreme direction of S_{th} for some values of t and h. According to Lemmas 5 and 6, Π^e can be obtained from (3.9) and (3.10). Equation (3.10) is equivalent to

$$[\alpha_1, \cdots, \alpha_k]\{B\bar{d}_t\bar{1} - B[\bar{d}_{t_1}, \cdots, \bar{d}_{t_{k-1}}, \bar{d}_t]\} = \bar{0}. \tag{a.10}$$

As mentioned before, $B = \begin{bmatrix} E_{r_1} \\ \cdots \\ E_{r_k} \end{bmatrix}$ where $r_1, \cdots, r_k \in \{1, \cdots, n\}$. Let D_t denote $D(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$. With little thought it can be seen that $B[d_1, \cdots, d_{k-1}, d_t] = D_t$. Next, it is shown that $V = \beta\bar{1}D_t^{-1}$ is a solution of (a.10). If $[\alpha_1, \cdots, \alpha_k]$ is substituted by V, the left side of (a.10) becomes

$$\beta\bar{1}D_t^{-1}B\bar{d}_t\bar{1} - \beta\bar{1}. \tag{a.11}$$

Let $\det M$ denote the determinant of a square matrix M. Notice that $B\bar{d}_t$ is the last column of D_t, $D_t^{-1} = D_t^*/(\det D_t)$ where D_t^* is the adjugate [18, p. 170]) matrix of D_t and $D_t^* B\bar{d}_t = [0, \cdots, 0, \det D_t]^T$. So (a.11) becomes

$$\beta\bar{1}\begin{bmatrix} \bar{0} \\ 1 \end{bmatrix}\bar{1} - \beta\bar{1} = \bar{0}.$$

Therefore, V is a solution of (a.10). By (3.9), $\Pi^e = [\alpha_1, \cdots, \alpha_k]B = VB = \Pi(t_1 \cdots t_{k-1} t / r_1 \cdots r_k)$. So $\Pi^e \in C_h^o$. By Lemmas 1–4, Π^e is one of the extreme directions. Therefore, $\Pi^e \in C_h^o$. □

List of Symbols

A: index constraint matrix; see Assumption 2.1.

A_i: row vector with n components; the ith row of matrix A.

a: number of rows of index constraint matrix A; see Assumption 2.1.

B: vector space; norm intersection; see Definition 2.2.

B: a matrix.

B_i: row vector with n components; the ith row of matrix B.

$\bar{b}$: size vector (column); see Assumption 2.1.

C^o: candidate set for (J, D) constructed by Procedure 3.1.

C_h^o: candidate set for (J, D) constructed by Procedure 5.1 where J is hyperparallelepiped.

D: dependence matrix with n rows and m columns; see Definition 2.1 (4).

$\bar{d}_i$: dependence (column) vector with n components; see Definition 2.1 (4).

$\dim\{\Omega\}$: the number of linearly independent vectors in the vector space (or set) Ω.

E_i: row vector with n components whose entries are zero except that the ith entry is one.

$\bar{e}$: column vector; extreme point of index set J.

F: set; boundary surface of J; see Definition 2.2.

f: objective function of Problem 2.1.

H_1: convex hull of all heads of projection vectors in (3.8b).

H_2: convex hull of all tails of projection vectors in (3.8b).

I: identity matrix.

$\mathbf{R}$: set of real numbers.

J: index set; see Definition 2.1 (1).

$\bar{j}$: column vector; index point; see Definition 2.1 (1).

m: number of dependence vectors in D; see Definition 2.1 (4).

N: vector space; norm space; see Definition 2.2

N: set of nonnegative integers.

N^+: set of positive integers.

n: number of components of index points in J; see Definition 2.1 (1).

P: set of projection vector; see Definition 2.6.

$\bar{p}$: projection vectors; see Definition 2.6.

Q: set of rational numbers.

q: number of projection vectors in P; see Definition 2.6.

R: convex hull of the index set J; see Assumption 2.1.

$\text{rank}(A)$: rank of matrix A.

S: the solution space of Problem 2.1.

S_{th}: subset of S where $\bar{d}_t$ is minimum and $\bar{p}_h$ is maximum; see (3.6).

$sp\{\bar{v}_1, \cdots, \bar{v}_k\}$: the vector space spanned by vector $\bar{v}_1, \cdots, \bar{v}_k$.

$\text{surf}\{A_{h_1}, \cdots, A_{h_k}\}$: boundary surface $\{\bar{x} : A_{h_i}\bar{x} = b_{h_i}, \bar{x} \in \mathbf{R}^n, i = 1, \cdots, k\}$; see Definition 2.2.

$\bar{x}$: a real column vector.

Z: set of integers.

Π: row vector; linear schedule vector; see Definition 2.5.

Π^o: row vector; optimal solution of Problem 2.1.

Π^e: row vector, the extreme direction satisfying (3.8).

Π_{th}: row vector; optimal solution of linear program (4.3).

σ: a schedule (function); see Definition 2.3.

σ_Π: linear schedule; see Definition 2.5.

$\varnothing$: empty set.

$\bar{1}$: a column or row vector whose entries are all 1.

$\bar{0}$: a column or row vector whose entries are all 0.

$|S|$: cardinality of set S.

REFERENCES

[1] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, July 1967.

[2] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. C-35, pp. 1–12, Jan. 1986.

[3] P. R. Cappello and K. Steiglitz, "Unifying VLSI array designs with geometric transformations," in *Proc. 1983 Int. Conf. Parallel Processing*, 1983, pp. 448–457.

[4] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proc. 11th Annu. Symp. Comput. Architecture*, 1984, pp. 208–214.

[5] S. K. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Stanford Univ., Stanford, CA, Oct. 1985.

[6] M. Chen, "A design methodology for synthesizing parallel algorithms and architectures," *J. Parallel Distributed Comput.*, pp. 461–491, Dec. 1986.

[7] J.-M. Delosme and I. C. F. Ipsen, "An illustration of a methodology for the construction of efficient systolic architectures in VLSI," in *Proc. Second Int. Symp. VLSI Technol. Syst. Appl.*, 1985, pp. 268–273.

[8] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[9] C. Guerra and R. Melhem, "Synthesizing non-uniform systolic designs," in *Proc. 1986 Int. Conf. Parallel Processing*, 1986, pp. 765–771.

[10] G.-J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 66–77, Jan. 1985.

[11] M. T. O'Keefe and J. A. B. Fortes, "A comparative study of two systematic design methodologies for systolic arrays," in *Proc. 1986 Int. Conf. Parallel Processing*, 1986, pp. 672–675.

[12] J. A. B. Fortes and F. Parisi-Presicce, "Optimal linear schedules for the parallel execution of algorithms," in *Proc. 1984 Int. Conf. Parallel Processing*, 1984, pp. 322–328.

[13] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[14] M. S. Bazaraa and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*. New York: Wiley, 1979.

[15] W. Shang and J. A. B. Fortes, "Independent partitioning of algorithms with uniform dependencies," in *Proc. 1988 Int. Conf. Parallel Processing*, vol. 2, *Software*, Aug. 1988, pp. 26–33.

[16] ——, "Time optimal linear schedules for algorithms with uniform dependencies," in *Proc. Int. Conf. Systolic Arrays*, May 1988, pp. 393–402.

[17] R. T. Rockafellar, *Convex Analysis*. Princeton, NJ: Princeton Univ. Press, 1970.

[18] G. Strang, *Linear Algebra and its Applications*. 2nd ed. New York: Academic, 1980.

[19] R. Cytron, "Doacross: Beyond vectorization for multiprocessors (extended abstract)," in *Proc. 1986 Int. Conf. Parallel Processing*, 1986, pp. 836–844.

[20] W. L. Miranker and W. M. Liniger, "Parallel methods for the numerical integration of ordinary differential equation," *Math. Comp.*, vol. 21, pp. 303–320, 1967.

[21] R. H. Kuhn, "Optimization and interconnection complexity for: parallel processors, single stage networks, and decision trees," Ph.D. dissertation, Dep. Comput. Sci., Rep. 80-1009, Univ. of Illinois, Urbana-Champaign, IL, 1980.

[22] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI array processors," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980, section 8.3.

[23] N. Weste, D. J. Burr, and B. D. Ackland, "Dynamic time warp pattern matching using an integrated multiprocessing array," *IEEE Trans. Comput.*, vol. C-23, pp. 731–744, Aug. 1983.

[24] D. A. Padua, "Multiprocessors: Discussion of theoretical and practical problems," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Rep. UIUCDCS-R-79-990, Nov. 1979.

[25] J.-K. Peir and R. Cytron, "Minimum distance: A method for partitioning recurrences for multiprocessors," in *Proc. 1987 Int. Conf. Parallel Processing*, 1987, pp. 217–225.

[26] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. Fifteenth Annu. ACM. SIGACT–SIGPLAN Symp. Principles Programming Languages*, Jan. 1988, pp. 319–329.

[27] B. Lisper, "Synthesis of synchronous systems by static scheduling in space-time," Ph.D. dissertation, TRITA-NA-8701, NADA, KTH, Stockholm, 1987.

[28] V. Van Dongen, personal communications.

[29] A. Schrijver, *Theory of Linear and Integer Programming*. New York: Wiley, 1986.

[30] W. Shang, "Scheduling, partitioning and mapping of uniform dependence algorithms on processor arrays," Ph.D. dissertation, Purdue Univ., W. Lafayette, IN 47907, May 1990.

[31] D. G. Luenberger, *Linear and Nonlinear Programming*. 2nd Ed. Reading, MA: Addison-Wesley, 1984.

Weijia Shang (S'90) received the B.S. degree in computer engineering and science from Changsha Institute of Technology, China, in 1982 and the M.S. degree in electrical engineering from Purdue University, West Lafayette, IN, in 1984.

She is currently a Ph.D. student in the School of Electrical Engineering, Purdue University, West Lafayette, IN. Her research interests include parallel processing, computer architecture, scheduling problems, and processor arrays.

Ms. Shang is a student member of the Association for Computing Machinery.

Jose A. B. Fortes (S'80–M'83) received the M.S. and Ph.D. degrees in electrical engineering from Colorado State University and the University of Southern California, respectively.

He is an Associate Professor at the School of Electrical Engineering of Purdue University. His technical interests are in the areas of parallel processing and fault-tolerant computing on which he has published more than 50 papers. He has been on the faculty of Purdue University, West Lafayette, IN, since 1984. From August of 1989 until July of 1990, he was on leave at the National Science Foundation where he was a Program Director for Microelectronic Systems Architecture.

Dr. Fortes is on the editorial boards of the *Journal of Parallel and Distributed Computing* and of the *Journal of VLSI Signal Processing* and co-chaired the program committee of the 1990 International Conference on Applications Specific Array Processors (ASAP'90).

REFERENCE NO. 4

Shang, W. and Fortes, J. A. B., "On the Independent Partitioning of Algorithms with Uniform Data Dependencies," *IEEE Transactions on Computers,* Volume 41, Number 2, February 1992, pp. 190-206.

Note - This paper presents a technique for optimal partition of algorithms into blocks of computations among which no communication is required. The same techniques can be used for the case when limited communication is possible. These techniques are potentially useful to reallocate computations in multiprocessor systems where failures result in diminished connectivity among processors.

Independent Partitioning of Algorithms with Uniform Dependencies

Weijia Shang, *Member, IEEE*, and Jose A. B. Fortes, *Member, IEEE*

Abstract— **An algorithm can be thought of as a set of indexed computations (index set) and if one computation uses data generated by another computation, this data dependence can be represented by the difference of their indexes (called dependence vector). Many important algorithms are characterized by the fact that data dependencies are uniform, i.e., the values of the dependence vectors are independent of the indexes of computations. An independent partition of the algorithm is such that there are no dependencies between computations that belong to different blocks of the partition. This paper considers uniform dependence algorithms with arbitrary index sets and proposes two computationally inexpensive methods to find their independent partitions. Each method has advantages over the other one for certain kinds of applications, and they both outperform previously proposed approaches in terms of computational complexity and/or optimality. Also, lower and upper bounds are given for the cardinality of maximal independent partitions. In MIMD systems, if different blocks of an independent partition are assigned to different processors, communications between processors will be minimized to zero. This is significant because the communications usually dominate the overhead in MIMD machines.**

Index Terms— **Data communication, independent algorithm partition, multiprocessor, nested loops, optimizing compiler, synchronization.**

I. Introduction

PARALLEL processing holds the potential for computational speeds that surpass by far those achievable by technological advances in sequential computers. This potential is predicated on two often conflicting assumptions, namely, that many computations can take place concurrently and that the time spent in data exchanges between these computations is small. In order to meet these assumptions, algorithms and/or programs must be partitioned into computational blocks that can execute in parallel and have communication requirements efficiently supported by the target parallel computer. Ideally, it may be desirable to identify, if at all possible, the independent computational blocks of a program, i.e., those that require no data communication between them. This paper describes two practical and computationally inexpensive approaches to achieve this goal. They are based on a sound mathematical framework which yields optimal results for a meaningful class of algorithms and they outperform approaches proposed in extant work.

The identification of a possible partition of an algorithm or program can be done by the user, by the analysis phase of an optimizing compiler, or by the machine at run time [4]. The techniques proposed in this paper, while usable by a patient and dedicated programmer, are best suited for an optimizing compiler. They address the specific problem of identifying *independent partitions* of an algorithm with goals that are similar to those of the earlier works of Padua [13] and Peir. Gajski and Cytron [17], [16], [15]. The focus of these efforts is on the optimization of programs consisting mainly of nested loops with regular data dependencies. The techniques proposed in those papers are intended to complement many other tools for the analysis and restructuring of sequential programs for execution in multiprocessing machines [1], [14], [25], [8], [18]. A related potential application of partitioning techniques is in the design of algorithmically specialized concurrent VLSI architectures [10].

In this paper, nested loop programs with regular data dependencies are modeled as *uniform dependence algorithms* which resemble the uniform recurrence equations considered in [7] and the linear recurrences of [15]. Data dependencies are represented as dependence vectors (with as many entries as the number of nested loops) that describe the distance between dependent computations in terms of loop indexes (the vectors are called dependence distance vectors in [15] and are also considered in [25] and [2] in a complemented form). Dependence vectors are collected in a matrix, the *dependence matrix*, which is used in this paper and in [13], [17], and [15] to identify independent partitions as briefly described in the following paragraphs.

The *greatest common divisor method* [13], [15] considers, for each row of the dependence matrix, the greatest common divisor of the entries in that row. The resulting greatest common divisors are used to partition the iteration space of the program (also called the index set) and the cardinality of the resulting partition is the product of the greatest common divisors. In addition, an "alignment" method is provided in [13] which allows in some cases the transformation of dependencies so that the values of the greatest common divisors are increased. This approach is simple and computationally inexpensive. For a given set of dependencies, it yields a unique independent partition which is not necessarily optimal. In some cases, when all of the greatest common divisors equal unity,

Manuscript received March 15, 1988; revised March 25, 1991. This work was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contracts 00014-85-k-0588 and 00014-88-k-0723.

W. Shang is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9103034.

the number of the blocks in the partition is one, i.e., the whole program.

In the *minimum distance method* [17], [15], the dependence matrix is transformed into an upper triangular matrix which is then used to identify an independent partition. For some algorithms the cardinality of the partition is the product of the diagonal elements of the upper triangular matrix. This approach yields partitions that are better than those obtained through the greatest common divisor method. However, the computational complexity of this method is higher (though affordable according to [15]) and the optimality is not always guaranteed.

Two approaches are proposed in this paper. In the first approach, called *partitioning vector approach*, a set of vectors (defined later in Section III) is derived from the dependence matrix. These vectors are used to find independent partitions of uniform dependence algorithms. The block to which a given computation belongs can be identified by simply computing the dot products of each of the vectors by the index of the computation. In the second approach, called *Smith normal form approach*, a matrix related to the Smith normal form of the dependence matrix is used to find independent partitions of uniform dependence algorithms. The block to which a given computation belongs can be identified by the product of that matrix and the index of the computation. Both approaches provide lower bounds and upper bounds on the cardinality of the resulting partitions. The first approach gives maximal partitions for a meaningful class of algorithms and the second approach yields maximal partitions for any algorithms with uniform dependence structure. Comparisons of these two approaches proposed in this paper and the minimum distance method are provided in Section VI.

The organization of this paper is as follows. Section II presents basic definitions and notation. Sections III and IV present the partitioning vector approach. In Section III, partitioning and separating vectors are defined and three types of independent algorithm partitions by these vectors are derived. In Section IV, a procedure to use these vectors to find an independent algorithm partition is presented and sufficient conditions for the resulting partition to be maximal are discussed. Section V introduces the notion of Smith normal form and presents a procedure to find independent partitions based on this notion. Section VI compares the methods proposed in this paper and the minimum distance method in [15]. Section VII summarizes conclusions and points out some future work.

II. Basic Definitions and Notation

Throughout this paper, *sets*, *matrices*, and *row vectors* are denoted by capital letters, column vectors are represented by lower-case symbols with an overbar and scalars correspond to lower-case letters. The *transposes* of a vector $\bar{v}$ and a matrix M are denoted $\bar{v}^T$ and M^T, respectively. The symbol E_i denotes the row vector whose entries are all zeros except that the ith entry is equal to unity. The vector $\bar{1}$ (or $\bar{0}$) denotes the row vector or column vector whose entries are all ones (or zeroes). The dimensions of $\bar{1}$ and $\bar{0}$ and whether they denote row or column vectors are implied by the context in which they are used. The vector space spanned by a set of vectors $W = \{\bar{v}_1, \bar{v}_2, \cdots, \bar{v}_k\}$ is denoted $sp\{\bar{v}_1, \bar{v}_2, \cdots, \bar{v}_k\} = sp\{W\}$. The symbol I denotes the identity matrix. The rank of a matrix A is denoted $\text{rank}(A)$ and the determinant of matrix A is represented by $\det A$. The set of real numbers and the set of integers are denoted $\mathbf{R}$ and Z, respectively. The set of nonnegative integers and the set of positive integers are denoted N and N^+, respectively. The empty set is denoted $\varnothing$ and the notation $A - B$ denotes the set $\{x : x \in A, x \notin B\}$ where A and B are sets. The notation $|P|$ means the cardinality of set P and $|c|$ represents the absolute value of scalar c. Let $a, b, c, d \in Z$ and $a > 0$, the notation $a \mid b$ means "a divides b", i.e., $b = ca$; and $b(\text{mod } a)$ denotes the modulo operation, i.e., $b(\text{mod } a) = d$ if and only if $a \mid (b - d)$, $0 \le d < a$. As a final remark, if the element a belongs to a set W, the notation $a \in W$ is used and this notation is also used to indicate that a column vector $\bar{m}_j$ (or a row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j \in M (M_i \in M)$ means $\bar{m}_j (M_i)$ is a column (row) vector of matrix M.

The algorithms of interest in this paper are the so-called uniform dependence algorithms defined as follows.

Definition 2.1 (Uniform dependence algorithm): A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\bar{j}) = f_{\bar{j}}(v(\bar{j} - \bar{d}_1), v(\bar{j} - \bar{d}_2), \cdots, v(\bar{j} - \bar{d}_m)) \qquad (2.1)$$

where

1) $\bar{j} \in J \subset Z^n$ is an index point (column vector), J is the index set of the algorithm and $n \in N^+$ is the number of components of $\bar{j}$;
2) $f_{\bar{j}}$ is the computation indexed by $\bar{j}$, i.e., a single-valued function computed "at point $\bar{j}$" in a *single unit of time*;
3) $v(\bar{j})$ is the value computed "at $\bar{j}$," i.e., the result of computing the right-hand side of (2.1) and
4) $\bar{d}_i \in Z^n, i = 1, \cdots, m, m \in N$ are *dependence vectors*, also called *dependencies*, which are constant (i.e., independent of $\bar{j} \in J$); the matrix $D = [\bar{d}_1, \cdots, \bar{d}_m]$ is called the *dependence matrix* and $\text{rank}(D) \le \min\{n, m\}$ is denoted by m'.

The class of uniform dependence algorithms is a simple extension of the class of computations described by uniform recurrence equations [7]. The main difference is that uniform dependence algorithms allow for different functions to be computed (in a unit of time) at different points of the index set. From a practical viewpoint, uniform dependence algorithms can be easily related to programs where 1) a single statement appears in the body of a multiply nested loop and 2) the indexes of the variable in the left-hand side of the statement differ by a constant from the corresponding indexes in all references to the same variable in the right-hand side. Alternative computations can occur in each iteration as a result of a single conditional statement as long as data dependencies do not change. Nested loop programs with multiple statements can also use the techniques of this paper together with the alignment method discussed in [13] and [15]. For the purpose of this paper, only structural information of the algorithm, i.e., the index set J and the dependence matrix D, is needed. Other information such as what computations occur at

different points and where and when input/output of variables takes place can be ignored. Therefore, a uniform dependence algorithm with index set J and dependence matrix D is hereon characterized simply by the pair (J, D). Also, as in Definition 2.1, the letters n, m, and m' always denote the dimension of index points in J, the number of dependence vectors and the rank of the dependence matrix D, respectively.

Definition 2.2 (Algorithm dependence graph and connectivity): The dependence graph of an algorithm (J, D) is the nondirected graph (J, E) where J is the set of nodes of the graph and $E = \{(\bar{j}', \bar{j}) : \bar{j} - \bar{j}' = \bar{d}_i \text{ or } \bar{j}' - \bar{j} = \bar{d}_i, \bar{d}_i \in D, \bar{j}', \bar{j} \in J\}$ is the set of edges. Two index points $\bar{j}, \bar{j}'$ are connected if there exist index points $\bar{j}_1, \cdots, \bar{j}_l \in J$ such that $(\bar{j}, \bar{j}_1), (\bar{j}_1, \bar{j}_2), \cdots, (\bar{j}_{l-1}, \bar{j}_l), (\bar{j}_l, \bar{j}') \in E$.

Definition 2.3 (Independent partition, maximal independent partition and partitionability): Given an algorithm (J, D) and the corresponding dependence graph (J, E), let $\rho = \{J_1, \cdots, J_q\}, q \in N^+$, be a partition of J. If for any arbitrary points $\bar{j}_1 \in J_i$ and $\bar{j}_2 \in J_l, i \neq l$ and $0 < i, l \leq q, (\bar{j}_1, \bar{j}_2) \notin E$, then ρ is an *independent partition* of the algorithm (J, D). The sets J_i, $i = 1, \cdots, q$, are called *blocks* of partition ρ. For an independent partition ρ, if any two arbitrary points $\bar{j}, \bar{j}' \in J_i, i = 1, \cdots, q$, are connected in the dependence graph, then ρ is the *maximal independent partition* of (J, D) and is denoted $\rho_{\max}$. The cardinality of the maximal independent partition $|\rho_{\max}|$ is referred to as the partitionability of the algorithm (J, D).

Informally, for the algorithm dependence graph, each index point corresponds to a node and if a computation depends directly on another one, then there is an edge between their index points. For Definition 2.3, an independent partition is such that there are no dependencies between computations which belong to different blocks of the partition. In graph theoretical terms, each block of an independent partition of (J, D) corresponds to a component of its dependence graph (J, E).

Generally speaking, the shape and the size of the index set influence the partitionability of the algorithm because of boundary conditions. Consider two algorithms (J, D) and (J, D') such that $D' = D, J = J' \bigcup \{\bar{j}\}$ and $\bar{j} \notin J'$, i.e., they differ only in the size of the index sets. The corresponding dependence graphs (J, E) and (J', E') can be such that $\bar{j}_1, \bar{j}_2 \in J'$ are not connected in (J', E') but are connected in (J, E) because it is possible that $E = E' \bigcup \{(\bar{j}_1, \bar{j}), (\bar{j}, \bar{j}_2)\}$. In other words, $\bar{j}_1$ and $\bar{j}_2$ can belong to different blocks of the maximal independent partition of (J', D') but belong to the same block of the maximal independent partition of (J, D). The following example illustrates this concept.

Example 2.1: Consider algorithms (J, D) and (J', D), where

$$D = \begin{bmatrix} 3 & 0 \\ -3 & 2 \end{bmatrix}, J = \{[j_1, j_2]^T : 0 \leq j_1, j_2 \leq s, s \in N^+\}$$

and

$$J' = \{\bar{j} : \begin{bmatrix} -1 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix} \bar{j} \leq \begin{bmatrix} 0 \\ 0 \\ s \end{bmatrix}, s \in N^+\}.$$

Fig. 1 shows the index sets J and J' where $s = 8$. These two algorithms have the same dependence matrix but different index sets. In J', point $[1, 1]^T$ is not connected to any other points because $[1, 1]^T \pm \bar{d}_i$, $i = 1, 2$, do not belong to J'. However, in J, it is connected to $[4, 1]^T \in J'$. End of example.

The dependence of the partitionability of an algorithm (J, D) on the shape and size of its index set J is a complicated issue and has practical implications. For example, in many programs, the loop bounds are not known at compile time and partitions must be identified which are independent of the size and shape of the index set and based solely on data dependencies. To concentrate on the relationship between the dependence structure and the partitionability of the algorithm, the following concepts are introduced.

Definition 2.4 (Pseudo-connectivity): Given an algorithm (J, D), two index points $\bar{j}, \bar{j}' \in J$ are pseudo-connected if there exists a vector $\bar{\lambda} \in Z^m$ such that $\bar{j} = \bar{j}' + D\bar{\lambda}$.

As an example of pseudo-connectivity, in algorithm (J', D) of Example 2.1, point $[1, 1]^T$ is pseudo-connected to $[4, 0]^T$ through point $[1, 3]^T \in (J - J')$.

Definition 2.5 (Pseudo-independent partition, maximal pseudo-independent partition and pseudo-partitionability): Given an algorithm (J, D), let $P = \{J_1, \cdots, J_q\}$ be a partition of J. If any two arbitrary points $\bar{j}_1 \in J_i \in P$ and $\bar{j}_2 \in J_l \in P, i \neq l$, are not pseudo-connected, then P is a pseudo-independent partition of the algorithm (J, D). If P is a pseudo-independent partition and any two arbitrary points $\bar{j}, \bar{j}' \in J_i$, $i = 1, \cdots, q$, are pseudo-connected, then P is the maximal pseudo-independent partition of (J, D) and is denoted $P_{\max}$. The cardinality of the maximal pseudo-independent partition $|P_{\max}|$ is referred to as the pseudo-partitionability of the the algorithm (J, D).

In many practical cases, e.g., when "while" loops are present in a program, it is also convenient to consider algorithms whose index sets are arbitrarily large along one or more dimensions. The general case, i.e., when this applies to all dimensions, is captured in the following definition and is also considered in this paper.

Definition 2.6 (Semi-infinite index set): An index set J is semi-infinite if it takes the following form:

$$J = \{\bar{j} = [j_1, \cdots, j_n]^T : 0 \leq j_i < \infty, J_i \in N, i = 1, \cdots, n\} \tag{2.2}$$

Example 2.2: The algorithm (J, D), where $D = \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}$ and $J = N^2$ is semi-infinite, i.e., $J = \{\bar{j} = [j_1, j_2]^T : 0 \leq j_1, \bar{j}_2 < \infty, j_1, j_2 \in N\}$. The index set J is partially shown in Fig. 2. The maximal partition $\rho_{\max} = \{J_1, J_2, J_3, J_4\}$ where $J_1 = \{[0, 0]^T\}, J_2 = \{[1, 0]^T\}, J_3 = \{[0, 1]^T, [2, 0]^T\}$ and $J_4 = \{\bar{j} : \bar{j} \in (J - \bigcup_{i=1}^{3} J_i)\}$. Index points $\bar{j}_1 = [0, 0]^T$ and $\bar{j}_2 = [0, 1]^T$ are not actually connected in the dependence graph of the algorithm. However, they are pseudo-connected by Definition 2.4 since $\bar{j}_2 = \bar{j}_1 + D\bar{\lambda}, \bar{\lambda} = [3, 2]^T$. Intuitively, $\bar{j}_1$ and $\bar{j}_2$ are connected through points $[2, -1]^T, [4, -2]^T, [6, -3]^T$ and $[3, -1]^T$ which are not in J. Partition $\rho_{\max}$ is not a pseudo-independent partition. Since $\det D = 1$, equation $D\bar{\lambda} = \bar{j} - \bar{j}'$ always has an integer solution for $\bar{\lambda}$. So any two arbitrary points in J are

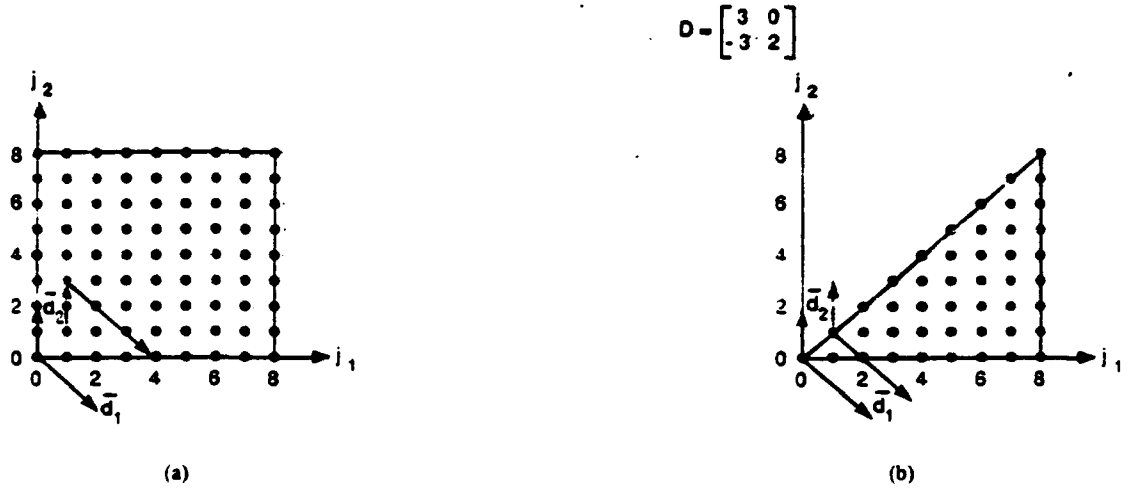


Fig. 1. In J', point $[1,1]^T$ is not connected to any other points. However, in J, it is connected to many other points such as point $[4,0]^T$. (a) Index set J. (b) Index set J'.

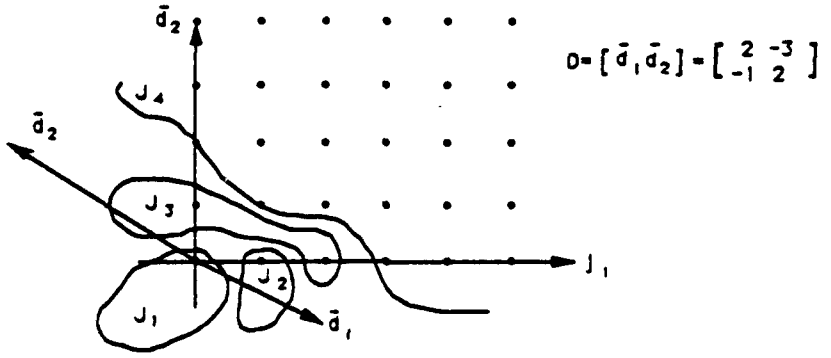


Fig. 2. The maximal independent partition of algorithm of Example 2.1 is $\rho_{max} = \{J_1, J_2, J_3, J_4\}$. However, there is only one block in the maximal pseudo-independent partition. Pictorially, only the connectivities of points near boundaries of J are influenced.

pseudo-connected. This implies that there is only one pseudo-independent partition $P = \{J\}$ which is also the maximal pseudo-independent partition. End of example.

At this point, some comments are in order. First, by Definitions 2.3 and 2.5, a pseudo-independent partition is also an independent partition regardless of the shape and size of the index set. However, an independent partition is not necessarily a pseudo-independent partition. This is due to the fact that $\bar{j}_1, \bar{j}_2 \in J$ are pseudo-connected if they are connected and the reverse is not necessarily true. Second, for practical purposes, it is sufficient and more efficient to identify pseudo-independent partitions than independent partitions for the reasons explained next. Blocks of independent partitions that are not blocks of a pseudo-independent partition and contain only a few index points such as J_1, J_2, and J_3 in Example 2.2 (hereon called boundary blocks) always occur at or near the boundaries of an index set. This can be shown for the general case when J is semi-infinite. In fact, according to Lemma 3 in [7], there exists always a point $\bar{p} = [p_1, p_2, \cdots, p_n]^T \in J$ such that if any arbitrary points $\bar{j} = [j_1, j_2, \cdots, j_n]^T \in J$ and $\bar{j}' = [j'_1, j'_2, \cdots, j'_n]^T \in J$ are beyond $\bar{p}$ (i.e., $j_i \geq p_i$ and $j'_i \geq p_i$, $i = 1, \cdots, n$), then $\bar{j}$ and $\bar{j}'$ are connected in the dependence graph if and only if they are pseudo-connected. Boundary blocks are typically such that their individual cardinalities are very small in relation to the sizes of the algorithm and pseudo-independent blocks. As a consequence, little additional speedup can result from executing boundary blocks concurrently with other blocks. Moreover, assigning small boundary blocks and other large pseudo-independent blocks to different processors of a multiprocessor can cause a nonbalanced load distribution and inefficient system operation. In addition, as pointed out before, when index sets are known only at run time, it is not possible to determine the boundary blocks. Finally, many algorithms are such that they have the same partitionability and pseudo-partitionability. For all of the above reasons, this paper considers hereon only the problem of identifying pseudo-independent partitions of an algorithm.

III. Partitioning Vector Approach—Basic Results

In this section and in Section IV the partitioning vector approach is presented. In this approach, independent algorithm partitions are determined by two types of vectors called partitioning vectors and separating vectors. Together with some auxiliary terminology they are introduced in Definitions 3.1 and 3.3. These definitions are followed by a theorem and an example which make clear the relation between these vectors and independent algorithm partitions.

Definition 3.1 (Partitioning vector, determining vector, equal partitioning vector and algorithm coefficient): Given an algorithm (J, D), $\Pi^1 = [\pi_1, \pi_2, \cdots, \pi_n] \in Z^{1\times n}$ is a *partitioning vector* of (J, D), if and only if it satisfies the following conditions.

1) $\gcd(\pi_1, \pi_2, \cdots, \pi_n) = 1$.[2]
2) There exists a set of $m' = \text{rank}(D)$ linearly independent dependence vectors $\bar{d}_{t_1}, \bar{d}_{t_2}, \cdots, \bar{d}_{t_{m'}}$ such that

$$\Pi\bar{d}_{t_1} = \cdots = \Pi\bar{d}_{t_{m'}} = disp\Pi > 0. \tag{3.1}$$

The dependence vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ are called the *determining vectors* of Π and $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}]$ is called *determining matrix* of Π. The positive integer $disp\Pi$ is called *displacement* of vector Π. If $\Pi\bar{d}_i \pmod{disp\Pi} = 0$, $i = 1, \cdots, m$, then Π is called an *equal partitioning vector* of (J, D). The constant $\alpha = \gcd(\Pi\bar{d}_1, \cdots, \Pi\bar{d}_m)$ is called the *algorithm coefficient*.

For a given partitioning vector the set of determining vectors is not necessarily unique and, therefore, $disp\Pi$ also might not be unique. However, given a partitioning vector and a set of determining vectors, $disp\Pi$ is unique. Therefore, whenever $disp\Pi$ is mentioned, it is associated with a particular set of determining vectors.

By Definition 3.1, if $m' = n$, then for each set of determining vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$, the corresponding partitioning vector Π is the unique solution that satisfies conditions 1 and 2 in Definition 3.1 and the following system of linear equations:

$$\begin{cases} \Pi(\bar{d}_{t_1} - \bar{d}_{t_2}) = 0 \\ \Pi(\bar{d}_{t_1} - \bar{d}_{t_3}) = 0 \\ \cdots \\ \Pi(\bar{d}_{t_1} - \bar{d}_{t_{m'}}) = 0. \end{cases} \tag{3.2}$$

When $m' < n$, the partitioning vector determined by m' linearly independent dependence vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ is not unique and, of course, it belongs to the solution space of (3.2). In the next section, a closed form expression is provided for a partitioning vector as a solution of (3.2).

A partitioning vector Π defines a set of hyperplanes $\Pi\bar{j} \pmod{\alpha} = c$, $c \in Z$, in the index space. Since an index point lies on only one of the hyperplanes, the index set J can be partitioned according to them, i.e., all points $\bar{j}$ lying on hyperplanes such that for a fixed c, $\Pi\bar{j} \pmod{\alpha} = c$, belong to the same block of the partition. The following definition states this concept formally.

Definition 3.2 (α-partition): Let Π be a partitioning vector and α be the algorithm coefficient for (J, D). The partition $P_\alpha = \{J_0, \cdots, J_{\alpha-1}\}$ where $J_i = \{\bar{j} : \bar{j} \in J, \Pi\bar{j} \pmod{\alpha} = i\}$, $i = 0, \cdots, \alpha - 1$, is called the *$\alpha$-partition* of (J, D).

Clearly, P_α is a partition of J and it will be shown in Theorem 3.1 that P_α is also a pseudo-independent partition. For the case where $m' < n$, i.e., $\text{rank}(D) < n$, a necessary condition for two index points $\bar{j}_1, \bar{j}_2 \in J$ to be pseudo-connected is that equation $D\bar{x} = (\bar{j}_1 - \bar{j}_2)$ has at least one real solution $\bar{x} \in \mathbf{R}^m$. This motivates the introduction of the following concepts. Let row vector Ψ_i be such that $\Psi_i D = \bar{0}$. Clearly, there are $n - m'$ linearly independent such vectors, denoted $\Psi_1, \cdots, \Psi_{n-m'}$, and they define a set of hyperplanes

$$\begin{bmatrix} \Psi_1 \\ \cdots \\ \Psi_{n-m'} \end{bmatrix} \bar{j} = \bar{y}, \bar{y} \in Z^{n-m'}, \tag{3.3}$$

in the index space. The index set J can be partitioned such that points lying on the same hyperplane belong to the same block of the partition. It will be clear later (Lemma 8.3) that if two index points $\bar{j}_1, \bar{j}_2 \in J$ lie on the same hyperplane defined by (3.3), then equation $D\bar{x} = (\bar{j}_1 - \bar{j}_2)$ has a solution. These concepts are formally defined as follows.

Definition 3.3 (Separating vector and separating matrix): Given an algorithm (J, D), $\Psi_i = [\psi_{i_1}, \cdots, \psi_{i_n}] \in Z^{1\times n}$ is a *separating vector* of (J, D) if and only if it satisfies the following conditions.

1) $\gcd(\psi_{i_1}, \cdots, \psi_{i_n}) = 1$
2) $\Psi_i D = \bar{0}$.

Let $\Psi_1, \cdots, \Psi_{n-m'}$ be all the linearly independent separating vectors; the matrix

$$\Psi = \begin{bmatrix} \Psi_1 \\ \cdots \\ \Psi_{n-m'} \end{bmatrix} \text{ is called } \textit{separating matrix}.$$

A set of $n - m'$ linearly independent separating vectors $\Psi_1, \cdots, \Psi_{n-m'}$ for algorithm (J, D) can be found by solving the equation in condition 2 of Definition 3.3. The following definition indicates how to use these separating vectors to construct a corresponding algorithm partition.

Definition 3.4 (Ψ-partition): Let Ψ be a separating matrix of algorithm (J, D). The partition $P_\Psi = \{J_{\bar{y}_1}, \cdots, J_{\bar{y}_\psi}\}$ of J is called the *Ψ-partition* of algorithm (J, D) if $J_{\bar{y}_i} = \{\bar{j} : \bar{j} \in J, \Psi\bar{j} = \bar{y}_i\}$, where $\bar{y}_i = [y_{1i}, \cdots, y_{(n-m')i}]^T \in Z^{(n-m')}$ is called the index of block $J_{\bar{y}_i}, i = 1, \cdots, \psi$.

Clearly, P_Ψ is a partition of J. If $m' = n$, then $P_\Psi = \{J\}$ is a trivial partition since the only separating vector is $\bar{0}$ in this case. As for P_α P_Ψ is actually pseudo-independent as shown later in Theorem 3.1.

Let $J_{\bar{y}} \in P_\Psi$ and consider the subalgorithm $(J_{\bar{y}}, D)$. Clearly, if $\alpha > 1$, subalgorithm $(J_{\bar{y}}, D)$ can be further partitioned by the partitioning vector Π. In other words, the index set J can be partitioned by a set of hyperplanes

$$\begin{bmatrix} \Pi\bar{j} \pmod{\alpha} \\ \Psi\bar{j} \end{bmatrix} = \begin{bmatrix} y_0 \\ \bar{y} \end{bmatrix},$$
$$y_0 \in \{0, 1, \cdots, \alpha - 1\} \text{ and } \bar{y} \in Z^{n-m'}, \tag{3.4}$$

and all points lying on the same hyperplane belong to the same block of the partition. This partition is formally stated next.

Definition 3.5 ($\alpha\Psi$-partition): Let Π be a partitioning vector and Ψ be a separating matrix of algorithm (J, D). The partition $P_{\alpha\Psi} = \{J_{\bar{y}_1}, \cdots, J_{\bar{y}_\nu}\}$ of index set J is called the *$\alpha\Psi$-partition* if $J_{\bar{y}_i} = \{\bar{j} : \bar{j} \in J, \begin{bmatrix} \Pi\bar{j} \pmod{\alpha} \\ \Psi\bar{j} \end{bmatrix} = \bar{y}_i\}$, where $\bar{y}_i = [y_{0i}, y_{1i}, \cdots, y_{(n-m')i}]^T \in Z^{n-m'+1}$ is called the index of block $J_{\bar{y}_i}$, $i = 1, \cdots, \nu$.

[1] Row vector Π should not be confused with the product notation $\prod$.

[2] $\gcd(a_1, \cdots, a_n)$ = the greatest common divisor of $a_1, \cdots, a_n$.

Partitioning vectors and separating vectors play a very important role in algorithm partition. The next theorem gives some of the motivation for the introduction of these concepts. More specifically, it provides sufficient conditions for two computations to belong to different blocks of an independent partition, in terms of those vectors and the index points associated with the computations. Moreover, it shows that α-partitions, Ψ-partitions, and $\alpha\Psi$-partitions are all pseudo-independent.

Theorem 3.1: Let Π be a partitioning vector, α be the algorithm coefficient, and Ψ be a separating matrix of algorithm (J, D), respectively. The following statements are true:

1) For any two arbitrary points $\bar{j}_1, \bar{j}_2 \in J$, if $\Pi\bar{j}_1 (\mathrm{mod}\ \alpha) \neq \Pi\bar{j}_2 (\mathrm{mod}\ \alpha)$ then they are not pseudo-connected. Therefore, P_α is a pseudo-independent partition of (J, D).
2) For any two arbitrary points $\bar{j}_1, \bar{j}_2 \in J$, if $\Psi\bar{j}_1 \neq \Psi\bar{j}_2$, then they are not pseudo-connected. Therefore, P_Ψ is a pseudo-independent partition of (J, D).
3) $P_{\alpha\Psi}$ is a pseudo-independent partition.

Proof: Provided in Appendix.

Corollary 3.1: If algorithm (J, D) has an equal partitioning vector Π, then $\bar{j}_1, \bar{j}_2 \in J$ are not pseudo-connected if $\Pi\bar{j}_1 (\mathrm{mod}\ disp\Pi) \neq \Pi\bar{j}_2 (\mathrm{mod}\ disp\Pi)$ or $\Psi\bar{j}_1 \neq \Psi\bar{j}_2$.

As a particular case of Theorem 3.1, Corollary 3.1 is obviously true. If algorithm (J, D) has an equal partitioning vector Π, then the algorithm coefficient $\alpha = disp\Pi$. By Theorem 3.1, Corollary 3.1 holds.

Example 3.1: Consider algorithm (J, D) where $J = \{[j_1, j_2]^T : 0 \leq j_1, j_2 \leq s, s \in N^+\}$ and $D = [\bar{d}]$ where $\bar{d} = [2, 2]^T$. Fig. 3 shows the index set J for $s = 4$. There is only one possible set of determining vectors $\{\bar{d}\}$. One of the partitioning vectors determined by $\bar{d}$ is $\Pi = [-1, 2]$. It follows that $disp\Pi = \Pi\bar{d} = 2$ and the algorithm coefficient $\alpha = 2$. Consider index points $\bar{j}_1 = [0, 0]^T$ and $\bar{j}_2 = [1, 0]^T$; since $\Pi\bar{j}_1 (\mathrm{mod}\ \alpha) = 0$ and $\Pi\bar{j}_2 (\mathrm{mod}\ \alpha) = 1$, by Theorem 3.1, they are not pseudo-connected. There is only one linearly independent separating vector $\Psi_1 = [1, -1]$ and a separating matrix is $\Psi = [1, -1]$. Again, consider index points $\bar{j}_1, \bar{j}_3 = [0, 1]^T$ for which $\Psi\bar{j}_1 = 0$ and $\Psi\bar{j}_3 = -1$. By Theorem 3.1, $\bar{j}_1$ and $\bar{j}_3$ are not pseudo-connected. In Fig. 3(a) and (b), hyperplanes $\Pi\bar{j} (\mathrm{mod}\ \alpha) = c_1$ and $\Psi\bar{j} = c_2, (c_1, c_2 \in Z)$, are drawn, respectively. All the points lying on the same hyperplane $\Pi\bar{j} (\mathrm{mod}\ \alpha) = c_1$ belong to the same block of the α-partition and all the points lying on the same hyperplane $\Psi\bar{j} = c_2$ belong to the same block of the Ψ-partition. Fig. 3 shows the α-partition, Ψ-partition, and $\alpha\Psi$-partition pictorially. Let $s = 3$, then $P_\alpha = \{J_0, J_1\}$ where $J_0 = \{[0, 0]^T, [0, 1]^T, [0, 2]^T, [0, 3]^T, [2, 0]^T, [2, 1]^T, [2, 2]^T, [2, 3]^T\}$ and $J_1 = \{[1, 0]^T, [1, 1]^T, [1, 2]^T, [1, 3]^T, [3, 0]^T, [3, 1]^T, [3, 2]^T, [3, 3]^T\}$. Also $P_\Psi = \{J_{[-3]}, \cdots, J_{[0]}, \cdots, J_{[3]}\}$ where $J_{[3]} = \{[3, 0]^T\}$, $J_{[2]} = \{[2, 0]^T, [3, 1]^T\}$, $J_{[1]} = \{[1, 0]^T, [2, 1]^T, [3, 2]^T\}$, $J_{[0]} = \{[0, 0]^T, [1, 1]^T, [2, 2]^T, [3, 3]^T\}$, $J_{[-1]} = \{[0, 1]^T, [1, 2]^T, [2, 3]^T\}$, $J_{[-2]} = \{[0, 2]^T, [1, 3]^T\}$ and $J_{[-3]} = \{[0, 3]^T\}$. The partition $P_{\alpha\Psi}$ can be obtained by intersecting $J_i \bigcap J_{[j]}$, $i = 0, 1$ and $j = -3, \cdots, 3$. Table I lists all blocks of $\alpha\Psi$-partition and their index points. $|P_{\alpha\Psi}| = 12 \leq \alpha|P_\Psi| = 14$. Clearly, P_Ψ, P_α, and $P_{\alpha\Psi}$ are pseudo-independent partitions. In Section IV, it is shown that the $\alpha\Psi$-partition is also the maximal pseudo-independent partition. End of example.

By Theorem 3.1, if there is at least one point $\bar{j} \in J$ such that $\Pi\bar{j} (\mathrm{mod}\ \alpha) = i$, then $J_i \in P_\alpha$ is not empty, i.e., $J_i \neq \emptyset$, $i = 0, \cdots, \alpha - 1$. Therefore, $|P_{\max}| \geq \alpha$. Intuitively, if J is large enough and *dense* (informally, an index set J is dense if any arbitrary point $\bar{j} \in Z^n$ that is inside the boundaries of J belongs to J), then for any arbitrary value of i, $0 \leq i < \alpha$ and $i \in Z$, there usually exists at least one index point $\bar{j}$ such that $\Pi\bar{j} (\mathrm{mod}\alpha) = i$. Therefore, it is reasonable to make the following assumption:

Assumption 3.1 (Index set): For an algorithm (J, D) under consideration in this paper, let Π be a partitioning vector and α be the algorithm coefficient. It is assumed that for any arbitrary value of $i \in Z$, $0 \leq i < \alpha$, there is at least one point $\bar{j} \in J$ such that $\Pi\bar{j} (\mathrm{mod}\ \alpha) = i$.

Corollary 3.2: Let α and P_α be the algorithm coefficient and the α-partition, respectively. Then $|P_\alpha| = \alpha$ under Assumption 3.1.

The next theorem shows that this is true if the index set J is defined by (2.2), i.e., $J = N^n$. Therefore, $|P_{\max}| \geq \alpha$ if J is semi-infinite.

Theorem 3.2: Let Π be a partitioning vector of (J, D) where J is defined by (2.2) and α be the algorithm coefficient. Then for any arbitrary value of $i \in Z$, $0 \leq i < \alpha$, there exists at least one index point $\bar{j} \in J$ such that $\Pi\bar{j} (\mathrm{mod}\alpha) = i$ and the pseudo-partitionability of (J, D) is greater than or equal to α, i.e., $|P_{\max}| \geq \alpha$.

Proof: Provided in Appendix.

IV. Partitioning Vector Approach—Procedure

In this section, Theorem 3.1 and other results and concepts introduced in Section III are used to prescribe a partitioning procedure. Afterwards, Section IV-A discusses how to find the partitioning vectors required by the procedure. Then Section IV-B characterizes algorithms for which the method yields the optimal partition and derives lower and upper bounds on the pseudo-partitionability of arbitrary uniform dependence algorithms. The independent partitioning procedure is as follows:

Procedure 4.1 (Finding $\alpha\Psi$-partition for algorithm (J, D) by partitioning vector approach):

Input: Algorithm (J, D).

Output: *$\alpha\Psi$-partition* $P_{\alpha\Psi}$ for algorithm (J, D).

Step 1: Select m' linearly independent dependence vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$, set $D_c = [\bar{d}_{t1}, \cdots, \bar{d}_{tm'}]$, find $T \in Z^{m' \times n}$ such that $\mathrm{rank}(TD_c) = m'$ and compute the corresponding partitioning vector Π according to Theorem 4.2 provided in Section IV-A. If $disp\Pi \neq |\det(TD_c)|$, then select another set of m' linearly independent dependence vectors and compute the corresponding partitioning vector until all distinct sets of m' linearly independent dependence vectors are considered. If a partitioning vector Π such that $disp\Pi = |\det(TD_c)|$ is not found, then select the partitioning vector Π such that $\frac{|\det(TD_c)|}{disp\Pi}$ is

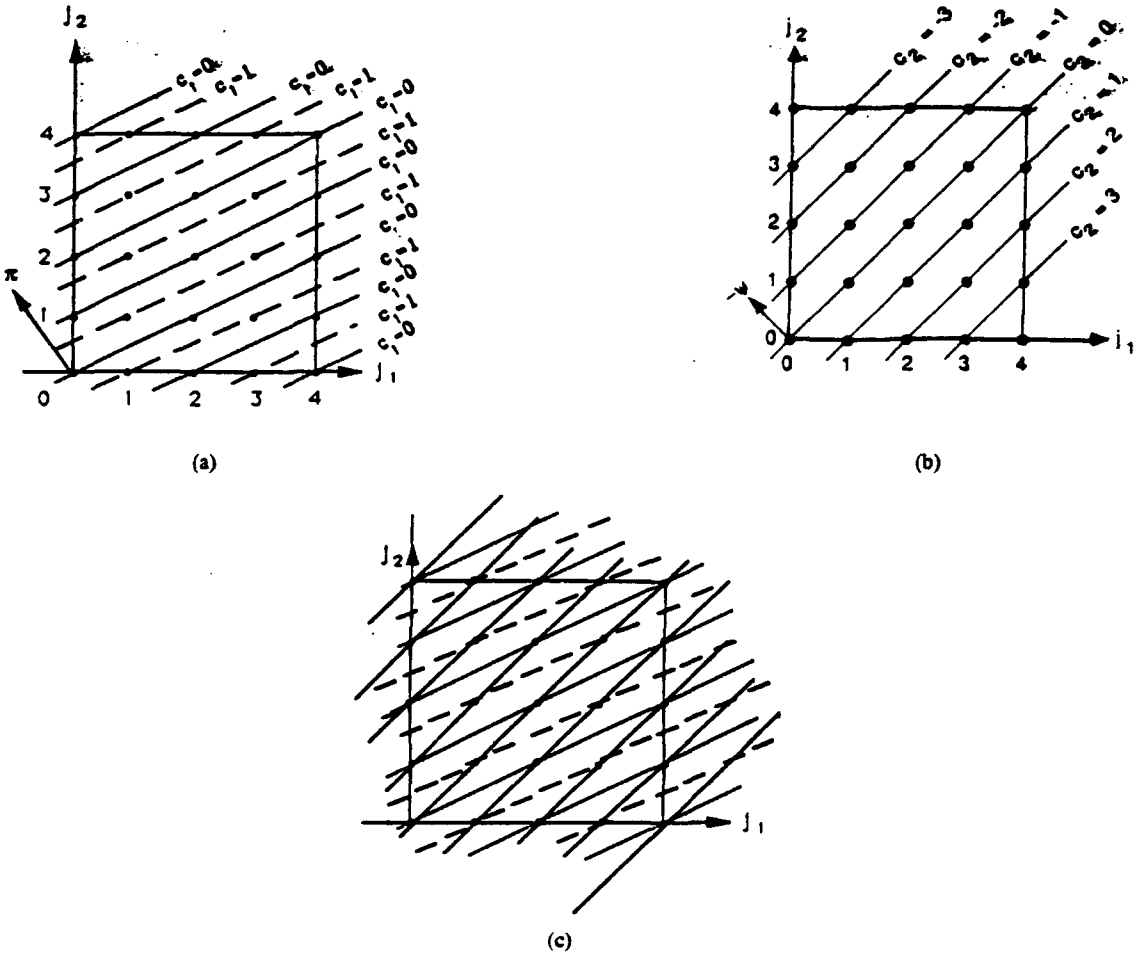


Fig. 3. Partitions of algorithm of Example 3.1 where $D = [2, 2]^T$, $\Pi = [-1, 2]$, and $\Psi = [1, -1]$. (a) α-partition: the hyperplanes are described by $\Pi\vec{j} \pmod 2 = c_1$. Points lying on dotted lines belong to block $J_0 \in P_\alpha$ and points lying on dashed lines belong to block $J_1 \in P_\alpha$. (b) Ψ-partition: the hyperplanes are described by $\Psi\vec{j} = c_2$. Points lying on hyperplane $\Psi\vec{j} = c_2$ belong to block $J_{[c_2]} \in P_\Psi$. (c) $\alpha\Psi$-partition: dotted and dashed lines specify the α-partition and solid lines specify the Ψ-partition.

TABLE I
LIST OF BLOCKS AND THEIR CORRESPONDING INDEX POINTS OF THE $\alpha\psi$-PARTITION OF ALGORITHM OF EXAMPLE 3.1

Block	$J\begin{bmatrix}0\\-3\end{bmatrix}$	$J\begin{bmatrix}1\\-3\end{bmatrix}$	$J\begin{bmatrix}0\\-2\end{bmatrix}$	$J\begin{bmatrix}1\\-2\end{bmatrix}$	$J\begin{bmatrix}0\\-1\end{bmatrix}$	$J\begin{bmatrix}1\\-1\end{bmatrix}$	$J\begin{bmatrix}0\\0\end{bmatrix}$	$J\begin{bmatrix}1\\0\end{bmatrix}$	$J\begin{bmatrix}0\\1\end{bmatrix}$	$J\begin{bmatrix}1\\1\end{bmatrix}$	$J\begin{bmatrix}0\\2\end{bmatrix}$	$J\begin{bmatrix}1\\2\end{bmatrix}$	$J\begin{bmatrix}0\\3\end{bmatrix}$	$J\begin{bmatrix}1\\3\end{bmatrix}$
Index Point	$\begin{bmatrix}0\\3\end{bmatrix}$	ϕ	$\begin{bmatrix}0\\2\end{bmatrix}$	$\begin{bmatrix}1\\3\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}2\\3\end{bmatrix}$	$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$ $\begin{bmatrix}2\\2\end{bmatrix}$	$\begin{bmatrix}1\\1\end{bmatrix}$ $\begin{bmatrix}3\\3\end{bmatrix}$	$\begin{bmatrix}2\\1\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}3\\2\end{bmatrix}$	$\begin{bmatrix}2\\0\end{bmatrix}$	$\begin{bmatrix}3\\1\end{bmatrix}$	ϕ	$\begin{bmatrix}3\\0\end{bmatrix}$

minimum. Then compute the algorithm coefficient $\alpha = \gcd(\Pi\vec{d}_1, \cdots \Pi\vec{d}_m)$.

Step 2: Obtain $n - m'$ linearly independent separating vectors $\Psi_1, \cdots, \Psi_{n-m'}$ by solving equation $\Psi_i D = \vec{0}$.

Set $\Psi = \begin{bmatrix}\Psi_1\\ \cdots \\ \Psi_{n-m'}\end{bmatrix}$

Step 3: For every index point $\vec{j} \in J$, if $\begin{bmatrix}\Pi\vec{j} \pmod\alpha\\ \Psi\vec{j}\end{bmatrix}$ $= \bar{y}_i = \begin{bmatrix}y_{0_i}\\ y_{1_i}\\ \cdots \\ y_{n-m'_i}\end{bmatrix}$, then assign $\vec{j}$ to $J_{\bar{y}_i}$, the block indexed by $\bar{y}_i$, i.e., $\vec{j} \in J_{\bar{y}_i}$.

Step 4: $P_{\alpha\Psi} = \{J_{\bar{y}_1}, \cdots, J_{\bar{y}_s}\}$; Stop. □

A. Finding a Partitioning Vector

This subsection provides in Theorem 4.2 a closed form

expression for the computation of partitioning vector Π, as required in Step 1 of the partitioning Procedure 4.1. In addition, an equal partitioning vector is preferred whenever it exists. This is because of the simple and regular mappings that result from equal partitioning vectors when time scheduling of computations is considered [21]. So, necessary and sufficient conditions are provided in Theorem 4.1 and Corollary 4.1 for the existence of this type of vectors for a given algorithm.

Theorem 4.1: An algorithm (J, D) has an equal partitioning vector if and only if there exists a set of m' linearly independent vectors $\bar{d}_{t1}, \bar{d}_{t2}, \cdots, \bar{d}_{tm'}$ such that

$$D = [\bar{d}_{t1}, \bar{d}_{t2}, \cdots, \bar{d}_{tm'}] \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m'1} & a_{m'2} & \cdots & a_{m'm} \end{bmatrix}, \quad a_{ij} \in \mathbb{R} \tag{4.1}$$

where $\sum_{i=1}^{m'} a_{ij}$ is an integer, $j = 1, \cdots, m$.

Proof: Provided in Appendix.

It is not easy to test whether a given algorithm has an equal partitioning vector using the condition in Theorem 4.1. The following corollary provides sufficient conditions which are easier to test.

Corollary 4.1: An algorithm (J, D) has an equal partitioning vector if it satisfies one of the following conditions:

1) $\text{rank}([\bar{d}_1 - \bar{d}_2, \bar{d}_1 - \bar{d}_3, \cdots, \bar{d}_1 - \bar{d}_m]) < \text{rank}(D)$.
2) There exists a set of m' linearly independent dependence vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ such that all dependence vectors can be expressed as an integer linear combination of $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$, i.e., $\bar{d}_j = \sum_{i=1}^{m'} a_{ij}\bar{d}_{t_i}, j = 1, \cdots, m$, where a_{ij}, $i = 1, \cdots, m'$ and $j = 1, \cdots, m$, are integer constants.

Proof: Provided in Appendix.

If algorithm (J, D) satisfies condition 1 in Corollary 4.1, then it has an equal partitioning vector Π such that $\Pi\bar{d}_1 = \cdots = \Pi\bar{d}_m = disp\Pi$. To see if a given algorithm satisfies condition 2 in Corollary 4.1, one has to see if (4.1) has an integer solution. This can be achieved by applying the necessary and sufficient conditions for a linear system of equations to have an integer solution provided in [19], [21].

Given m' linearly independent vectors $\bar{d}_{t1}, \cdots, \bar{d}_{tm'}$, the corresponding partition vector Π belongs to the solution space of (3.2). In [3], a closed form expression for a partitioning vector which is determined by $\bar{d}_{t1}, \cdots, \bar{d}_{tm'}$ is given. This result is restated as Theorem 4.2 as follows.

Theorem 4.2 [3]: Let $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ be linearly independent, consider matrix $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}]$ and let $T \in Z^{m' \times n}$ be such that $\text{rank}(TD_c) = m'$. Then $\Pi = \beta\bar{1}(TD_c)^{-1}T$ is a partitioning vector determined by $\bar{d}_{t1}, \cdots, \bar{d}_{tm'}$ and $disp\Pi = \beta$, where $\beta \in N^+$ is such that $\Pi \in Z^{1\times n}$ and the greatest common divisor of the n components of Π is equal to one.

Notice that matrix $T \in Z^{m' \times n}$ such that $\text{rank}(TD_c) = m'$ always exists. Because $\text{rank}(D_c) = m'$, there are m' linearly independent rows in D_c. Suppose rows $r_1, \cdots, r_{m'}$ are linearly independent. If $T = \begin{bmatrix} E_{r1} \\ \cdots \\ E_{rm'} \end{bmatrix}$ where $E_{r1}, \cdots, E_{rm'}$ are as defined in the beginning of Section II, then $\text{rank}(TD_c) = m'$. In other words, the result of multiplying D_c by T is a square submatrix of D that contains exactly m' linearly independent rows of the m' linearly independent columns of D. If $m' = n$, then $T = I$, the identity matrix, and $\Pi = \beta\bar{1}D_c^{-1}$. The essence of the proof is as follows [3]. Because $\beta\bar{1}(TD_c)^{-1}TD_c = \beta\bar{1}$, vector $\beta\bar{1}(TD_c)^{-1}T$ satisfies (3.1) and meets conditions 1 and 2 in Definition 3.1 by the meaning of the constant β; so $\Pi = \beta\bar{1}(TD_c)^{-1}T$ is a partitioning vector determined by $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ and $disp\Pi = \beta > 0$.

B. Sufficient Conditions for Optimality

Theorem 3.1 provides a necessary condition for two index points in J to be pseudo-connected. Next it is shown in Theorem 4.3 and 4.3a that when the dependence matrix D satisfies certain constraints, this condition becomes sufficient. The implication of this result is that the partition $P_{\alpha\Psi}$ obtained by Procedure 4.1 is maximal. In order to motivate and facilitate the understanding of the main results of this section, first, a special case is discussed in Theorem 4.3 where $m' = n$, i.e, $\text{rank}(D) = n$. In this case, the Ψ-partition is trivial, i.e., $P_\Psi = \{J\}$. Therefore, by Theorem 3.1, the necessary condition for two index points $\bar{j}_1, \bar{j}_2 \in J$ to be pseudo-connected is $\Pi\bar{j}_1 (\text{mod}\alpha) = \Pi\bar{j}_2 (\text{mod}\alpha)$.

Theorem 4.3: Let $m' = n$. Π be a partitioning vector of algorithm (J, D) determined by $\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}$, $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}]$ and α be the algorithm coefficient. If $|\det D_c| = disp\Pi$, then

1) two index points $\bar{j}_1, \bar{j}_2 \in J$ are pseudo-connected if and only if $\Pi\bar{j}_1 (\text{mod } \alpha) = \Pi\bar{j}_2 (\text{mod } \alpha)$;
2) the α-partition is the maximal pseudo-independent partition of (J, D), i.e., $P_{\max} = P_\alpha$, and $|P_{\max}| = \alpha$.

Proof: Provided in Appendix.

In this case, Procedure 4.1 becomes very simple. Since $\text{rank}(D) = n$, there is only one trivial separating vector 0 and therefore, $P_\Psi = \{J\}$. So Step 3 in Procedure 4.1 can be skipped. When Π is an equal partitioning vector, then $\Pi\bar{d}_i (\text{mod } disp\Pi) = 0$, $i = 1, \cdots, m$. So $\alpha = disp\Pi = |\det D_c|$. This fact is summarized as Corollary 4.2 as follows.

Corollary 4.2: Let $m' = n$, Π be an equal partitioning vector of algorithm (J, D) determined by $\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}$ and $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}]$. If $|\det D_c| = disp\Pi$, then the pseudo-partitionability of (J, D) is equal to the absolute value of the determinant of matrix D_c, i.e., $|P_{\max}| = |\det D_c|$.

The meaning of Corollary 4.2 is as follows. For a class of algorithms, the number of blocks in the maximal pseudo-independent partition is equal to $|\det D_c|$, the absolute value of the determinant of a submatrix of the dependence matrix D. If the algorithm is to be executed by clusters of processors with limited intercluster communication capabilities then the number of clusters to be used should be directly related and perhaps equal to the cardinality of the pseudo-independent partition. In such MIMD systems, $|\det D_c|$ is a direct indication of how many clusters can be used to execute the algorithm.

To find the necessary and sufficient conditions for two points $\bar{j}_1, \bar{j}_2 \in J$ to be pseudo-connected in general case, the approach used here is as follows. First, a subalgorithm (J_n, D) where $J_n \in P_\Psi$ is considered and the necessary

and sufficient conditions for two points $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ to be pseudo-connected are derived. To achieve this, the algorithm $(J_{\bar{y}}, D)$ is transformed, by a linear mapping T, into another algorithm $(T(J_{\bar{y}}), T(D))$ where the dimension of the index points is m' and there are m' linearly independent dependence vectors. Then Theorem 4.3 is applied to find these necessary and sufficient conditions for algorithm $(T(J_{\bar{y}}), T(D))$. Then it is shown that the mapping T is bijective and algorithms $(J_{\bar{y}}, D)$ and $(T(J_{\bar{y}}), T(D))$ are equivalent in the sense that $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ are pseudo-connected in algorithm $(J_{\bar{y}}, D)$ if and only if $T(\bar{j}_1), T(\bar{j}_2)$ are pseudo-connected in algorithm $(T(J_{\bar{y}}), T(D))$. So these necessary and sufficient conditions for algorithm $(T(J_{\bar{y}}), T(D))$ are actually valid for algorithm $(J_{\bar{y}}, D)$.

Theorem 4.3a: Consider algorithm (J, D), let $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ be linearly independent, $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}], T \in Z^{m' \times n}$ be such that $\text{rank}(TD_c) = m', \Pi = disp\Pi\bar{1}(TD_c)^{-1}T$ be the partitioning vector determined by $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}, \alpha$ be the algorithm coefficient and Ψ be a separating matrix. If $|\det(TD_c)| = disp\Pi$, then

1) two points $\bar{j}_1, \bar{j}_2 \in J$ are pseudo-connected if and only if $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod}\alpha) = 0$ and $\Psi\bar{j}_1 = \Psi\bar{j}_2$;
2) the $\alpha\Psi$-partition is the maximal pseudo-independent partition of (J, D), i.e., $P_{\max} = P_{\alpha\Psi}$.
3) $|P_\Psi| \le \prod_{i=1}^{n-m'}(\gamma_i + 1)$, where $\gamma_i = \max\{\Psi_i(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}, i = 1, \cdots, n - m'$, and $\alpha \le |P_{\max}| \le \alpha|P_\Psi|$.

Proof: Provided in Appendix.

If the cardinalities of the α-partitions of algorithms $(J_{\bar{y}}, D)$, where $J_{\bar{y}_i} \in P_\Psi$, $i = 1, \cdots, \psi$, are all equal to α, then $|P_{\max}| = \alpha|P_\Psi|$. However, for some block $J_{\bar{y}} \in P_\Psi$, the cardinality of its α-partition might be less than α because for some value of $i \in Z$, $0 \le i < \alpha$, there might not exist any index point $\bar{j} \in J_{\bar{y}}$ such that $\Pi\bar{j}(\text{mod } \alpha) = i$. This phenomenon is illustrated in the following example.

Example 4.1: Consider the algorithm of Example 3.1 with $s = 3$. There is only one set of determining vectors $\{\bar{d}\}$ and $D_c = D$. If $T = [-1, 2]$, then $TD_c = [2]$. According to Theorem 4.2, $\Pi = 2\bar{1}[2]^{-1}[-1.2] = [-1, 2]$ and $disp\Pi = 2 = \det(TD_c)$. As in Example 3.1, the separating matrix $\Psi = [1, -1]$. To illustrate Theorem 4.3a 1), consider points $\bar{j}_1 = [0.0]^T$ and $\bar{j}_2 = [2, 2]^T$. Because $\Pi\bar{j}_1(\text{mod } \alpha) = \Pi\bar{j}_2(\text{mod } \alpha)$ and $\Psi\bar{j}_1 = \Psi\bar{j}_2$, by Theorem 4.3a, they are pseudo-connected. Due to the fact that $disp\Pi = \det(TD_c)$, by Theorem 4.3a, $P_{\alpha\Psi}$ is the maximal pseudo-independent partition. Consider $J_{[3]} \in P_\Psi$, i.e., the block whose points $\bar{j}$ are such that $\Psi\bar{j} = 3$. $J_{[3]} = \{[3, 0]^T\}$ as found in Example 3.1. There does not exist any index point $\bar{j} \in J_{[3]}$ such that $\Pi\bar{j}(\text{mod } \alpha) = 0$. This illustrates the explanation before this example. By Theorem 4.3a 3), $|P_\Psi| \le \gamma + 1 = 7$, where $\gamma = 3 - (-3) = 6$ and $|P_{\max}| = 12 \le \alpha|P_\Psi| \le 14$. End of example.

The complexity of Procedure 4.1 is a linear function of the cardinality of the index set. Step 1 computes the partitioning vector. The complexity of the product of the number of memory locations (space) and the number of the operations (time) is bounded by $O(\binom{m}{m'}n^5)$. For step 2 the space×time complexity is bounded by $O((n - m')n^5)$. Step 3 needs at most $O(\min\{n, n - m' + 1\}n|J|)$ operations. The total complexity is bounded above by $O(\binom{m}{m'}n^5) + O((n - m')n^5) + O(\min\{n, n - m' + 1\}n|J|)$. When $m = m' = n$, the complexity is bounded above by $O(n^5) + O(n|J|)$.

V. Smith Normal Form Approach

The partitioning vector approach is simple and the partitioning vector and separating vectors are easy to compute. However, when the algorithm does not satisfy the condition in Theorem 4.3a, the optimality is not always guaranteed. This section discusses the *Smith normal form approach* which yields maximal pseudo-independent partitions for any arbitrary uniform dependence algorithms. This approach uses the Smith normal form of the dependence matrix D which is introduced in Theorem 5.1. This theorem is followed by the definitions of the partitioning matrix and the displacement vector. These concepts are then used to define a partition of index set J which is also the maximal pseudo-independent partition of the algorithm. Then a procedure is presented which constructs the maximal pseudo-independent partition of a given algorithm. Complexity of the procedure is also discussed.

Theorem 5.1 (Smith normal form) [19, p. 50]: Given a matrix $D \in Z^{n \times m}$, there exist two unimodular[3] matrices $U \in Z^{n \times n}$ and $V \in Z^{m \times m}$ such that

$$UDV = S = \begin{bmatrix} s_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & s_{m'} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Matrix S is called the *Smith normal form* (abbreviated SNF) of matrix D, it is unique, $s_1, \cdots, s_{m'}$ are positive integers, $s_1, |s_2| \cdots |s_{m'}, \prod_{i=1}^{k} s_i$, $k = 1, \cdots, m'$, is the greatest common divisor of subdeterminants of order k of the dependence matrix D and m' is the rank of the dependence matrix D. □

More details and explanations about SNF can be found in [19, p. 50] and [24]. The following example illustrates these concepts just introduced.

Example 5.1: Consider the algorithm (J, D) studied in [15] for which $D = \begin{bmatrix} 1 & 2 & 0 \\ -2 & 4 & 4 \\ 4 & -1 & 2 \end{bmatrix}$ and $J = \{[j_1, j_2, j_3]^T : 1 \le j_i \le 16, j_i \in N^+, i = 1, 2, 3\}$. The matrices U, S, and V are as follows.

$$U = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -1 & -1 \\ -14 & 9 & 8 \end{bmatrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 52 \end{bmatrix} \quad \begin{bmatrix} 1 & -2 & -12 \\ 0 & 1 & 6 \\ 0 & 0 & 1 \end{bmatrix}$$

It is easy to verify that $UDV = S$ and U and V are unimodular. End of example.

[3] A nonsingular matrix is unimodular if its elements are integral and its determinant is ±1.

Definition 5.1 (Partitioning matrix and displacement vector): Given an algorithm (J, D), the matrix U such that $UDV = S$ is the SNF of D is called *partitioning matrix* of (J, D). Given that $s_1, \cdots, s_{m'}$ are the nonzero diagonal elements of S, the vector $\bar{s} = [s_1, \cdots, s_{m'}, \infty, \cdots, \infty]^T \in N^n$ is called *displacement vector* of (J, D).

Definition 5.2 (U-partition): Let U be a partitioning matrix of algorithm (J, D); the partition $P_U = \{J_{\bar{y}}, \cdots, J_{\bar{y}_\mu}\}$ of index set J is called the *U-partition* of algorithm (J, D) if $J_{\bar{y}_i} = \{\bar{j} : U\bar{j} (\text{mod } \bar{s}) = \bar{y}_i, \bar{j} \in J\}$,[4] where $\bar{y}_i \in Z^n$ is called the index of block $J_{\bar{y}_i}$, $i = 1, \cdots, \mu$.

Example 5.2: Consider the algorithm of Example 5.1. U is a partitioning matrix; $\bar{s} = [1, 1, 52]^T$ is the displacement vector; $P_U = \{J_{[0,0,0]^T}, \cdots, J_{[0,0,51]^T}\}$ is the U-partition where $J_{[0,0,i]^T} = \{\bar{j} : U\bar{j} (\text{mod} \bar{s}) = [0, 0, i]^T, \bar{j} \in J\}, i = 0, \cdots, 51$. End of Example.

It is clear that P_U is a partition of the index set J because for each $\bar{j} \in J, U\bar{j} (\text{mod} \bar{s})$ is unique. Actually, P_U is a pseudo-independent partition. To show the pseudo-independence of U-partition, the following lemma is introduced first and followed by a theorem.

Lemma 5.1: Given algorithm (J, D), let $\bar{j}_1, \bar{j}_2 \in J$ and $\bar{s}$ be the displacement vector, then $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected if and only if $U\bar{j}_1 (\text{mod } \bar{s}) = U\bar{j}_2 (\text{mod } \bar{s})$.

Proof: Provided in Appendix.

Theorem 5.2: Given an algorithm (J, D), let U_i be the ith row of the partitioning matrix U, $i = 1, \cdots, n$, $\delta_{iu} = \max\{U_i\bar{j} : \bar{j} \in J\}, \delta_{il} = \min\{U_i\bar{j} : \bar{j} \in J\}$ and $\gamma_i = \delta_{iu} - \delta_{il} + 1, i = m' + 1, \cdots, n$. The following statements are true:

1) The U-partition is the maximal pseudo-independent partition, i.e., $P_U = P_{\max}$.
2) The pseudo-partitionability is bounded above by $\prod_{k=1}^{m'} s_k \prod_{i=m'+1}^{n} \gamma_i$, i.e., $|P_{\max}| = |P_U| \leq \prod_{k=1}^{m'} s_k \prod_{i=m'+1}^{n} \gamma_i$.

Proof: Provided in Appendix.

For every vector $\bar{y} = [y_1, \cdots, y_n]^T, 0 \leq y_i < s_i, i = 1, \cdots, m', \delta_{il} \leq y_k \leq \delta_{iu}, k = m' + 1, \cdots, n$, if there exists at least one index point $\bar{j} \in J$ such that $\bar{j} \in J_{\bar{y}}$, then $J_{\bar{y}} \neq \emptyset$ and $|P_U| = \prod_{k=1}^{m'} s_k \prod_{i=m'+1}^{n} \gamma_i$. Because $\det U = \pm 1$, for each such vector $\bar{y}$, there always exists an integer vector $\bar{j}$ such that $U\bar{j} (\text{mod} \bar{s}) = \bar{y}$. Therefore, it is reasonable to assume that $|P_U| = \prod_{k=1}^{m'} s_k \prod_{i=m'+1}^{n} \gamma_i$. In particular, for algorithms whose index set $J = Z^n$, this assumption in true. For the special case where $m = m' = n$, by Theorem 5.1, $\prod_{i=1}^{n} s_i = |\det D|$. If for each vector $\bar{y} = [y_1, \cdots, y_n]^T, 0 \leq y_i < s_i, i = 1, \cdots, n$, there is at least one index point $\bar{j} \in J$ such that $\bar{j} \in J_{\bar{y}}$, then the following corollary is true.

Corollary 5.1: Given algorithm (J, D), let $m = m' = n$ and $\bar{s} = [s_1, \cdots, s_n]^T$ be the displacement vector. Then $|P_{\max}| = \prod_{i=1}^{n} s_i = |\det D|$, i.e., the pseudo-partitionability of algorithm (J, D) is equal to the absolute value of the determinant of dependence matrix D.

[4] Let $\bar{\lambda} = [\lambda_1, \cdots, \lambda_n]^T \in Z^n$ and $\bar{s} = [s_1, \cdots, s_{m'}, \infty, \cdots, \infty]^T$ with $s_i > 0$, $i = 1, \cdots, m'$. The notation $\lambda (\text{mod } \bar{s})$ denotes the vector $[\lambda_1 (\text{mod } s_1), \cdots, \lambda_{m'} (\text{mod } s_{m'}), \lambda_{m'+1}, \cdots, \lambda_n]^T$.

Procedure 5.1 (Finding the maximal pseudo-independent partition by SNF approach):

Input: Algorithm (J, D).

Output: U-partition P_U of algorithm (J, D).

Step 1: Find a partitioning matrix U and the displacement vector $\bar{s}$.

Step 2: For every index point $\bar{j} \in J$, compute $U\bar{j} (\text{mod } \bar{s}) = \bar{y}$ and assign j to $J_{\bar{y}}$, the block indexed by $\bar{y}$.

Step 3: $P_U = \{J_{\bar{y}_1}, \cdots, J_{\bar{y}_\mu}\}$. Stop. □

The complexity of Procedure 5.1 is a linear function of the cardinality of the index set J. i.e., the number of computations of the algorithm. Also, it is a polynomial function of n and m, the number of components of index vectors and the number of the dependence vectors, respectively. To construct the displacement vector $\bar{s}$, the SNF of matrix D is needed. In [6], a polynomial algorithm is proposed to find the SNF of any arbitrary matrix $A \in Z^{n \times m}$ and the corresponding left and right multipliers U and V such that $UAV = \text{SNF}$. The complexity of the product of the number of memory locations (space) and the number of operations (time) of this algorithm is $O((\max\{n, m\})^{10})$ [6]. So, the space x time complexity of Step 1 is bounded above by $O((\max\{n, m\})^{10})$. For Step 2, at most $O(|J|n^2)$ operations are needed to compute the value of $U_{\bar{y}}$ for every index point $\bar{y}$. Therefore, the total complexity is bounded above by $O((\max\{n, m\})^{10}) + O(|J|n^2)$. When $m = m' = n$, the complexity is bounded above by $O(n^{10}) + O(n^2|J|)$.

Example 5.3: Consider the matrix multiplication algorithm

$$c_{i,j}^0 = 0$$
$$c_{i,j}^k = c_{i,j}^{k-1} + a_{i,k} \cdot b_{k,j}, \quad k = 1, \cdots, s$$
$$i, j = 1, \cdots, s.$$

The dependence matrix and the index set are as follows:

$$D = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad J = \left\{ \begin{bmatrix} i \\ j \\ k \end{bmatrix} : 1 \leq i, j, k \leq s. \begin{bmatrix} i \\ j \\ k \end{bmatrix} \in Z^3 \right\}. \tag{5.1}$$

Notice that the dependence matrix above only considers the data dependence caused by c. If data broadcasting is not allowed or the dependence caused by data reads is considered, then the dependence matrix for the matrix multiplication algorithm is $D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ [11]. Dependences $[1, 0, 0]^T$ and $[0, 1, 0]^T$ are to avoid broadcasting data b_{ij} and a_{ij}, respectively.

For the dependence matrix D in (5.1), matrices U, V, and S and the displacement vector $\bar{s}$ are as follows:

$$U = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad V = [1],$$

$$S = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \bar{s} = [\ 1 \quad \infty \quad \infty\ ]^T.$$

If $s = 3$, according to Definition 5.2, the U-partition is $P_U = \{J_{[0,j,i]^T} : i, j = 1, 2, 3\}$ where $J_{[0,j,i]^T} =$

$\{[i,j,1]^T, [i,j,2]^T, [i,j,3]^T\}$. There are nine blocks in this maximal partition ($|P_U| = 9$). Intuitively, each block computes datum c_{ij}. If multiple copies of data a_{ij} and b_{ij} can be provided, then there is no communication/dependence among different blocks. Therefore, this partition is independent and also maximal. End of example.

VI. Comparative Evaluation of Partitioning Procedures

This section starts by discussing some of the advantages of the methods proposed in this paper over the earlier pioneering method reported in [15] and [17]. In Section VI-B the two approaches proposed in this paper are compared.

A. Minimum Distance Method

In the minimum distance method (abbreviated MDM) [15], [17], an elegant idea is used which consists of using a linear mapping to transform the dependence matrix D into an upper triangular matrix denoted D^t in [15]. These two dependence matrices are equivalent in the sense that each dependence vector in D^t, the upper triangular matrix, is a linear integer combination of the dependence vectors in D and vice versa. A set of initial points, each of which corresponds to a block in the resulting partition, is identified by D^t and the cardinality of the partition is the product of the diagonal elements of D^t. An independent partition is implicitly expressed by D^t and a set of initial points. "Independent partition" in [15] is used to denote the same as "pseudo-independent partition" in this paper.

The MDM finds the maximal pseudo-independent partition for a class of algorithms which is more restricted than the methods proposed in this paper. For the case where $m' = m = n$, the MDM generates the maximal pseudo-independent partition and for the case where $m' = m < n$, it generates an independent partition that may not be maximal. In [17], an algorithm to generate initial points is presented for this case. However, its complexity and optimality are not clear. Moreover, only index sets of the form $J = \{[j_1, \cdots, j_n]^T : a_i \le j_i \le b_i, i = 1, \cdots, n\}$ (not necessarily dense) are considered; otherwise, the initial points are not easy to identify.

Compared with the partitioning vector approach (abbreviated PVA) proposed in this paper, the MDM has the following disadvantages. First, in the MDM, partitions are expressed implicitly in terms of the upper triangular matrix and a set of initial points. According to [15], to find the upper triangular matrix, it is necessary to solve n integer programming problems with m variables which are NP-complete, where n, m are the number of dimensions of the index points and the number of dependence vectors, respectively. This is expensive although it is affordable when n, m are small. In the PVA, partitions are expressed explicitly in terms of the partitioning vectors and separating vectors. To obtain these vectors, the dominating computations required are to find partitioning vectors, i.e., consider at most all possible combinations of m' vectors from the m dependence vectors and compute $disp\Pi\bar{1}(TD_c)^{-1}T$. The complexity of the execution time of Procedure 4.1 is bounded above by $O(\binom{m}{m'}n^3)$.

Second, as mentioned above, in the MDM, blocks of the resulting partition are implicitly expressed in terms of the upper triangular matrix and a set of initial points. Although the serial loops in the original program can be transformed into parallel loops by the upper triangular matrix, it is costly to know which block a given index point belongs to. If the notation in [15] is used here, given an index point $X \in Z^{1\times n}$, then one way to see which block it belongs to is to see if equation $X = X_{i0} + AD^t$ has an integer solution $A \in Z^{1\times n}$, where X_{i0} is an initial point belonging to block i. If it has, then X belongs to block i. If it does not, then another initial point X_{j_0} belonging to block $j, j \neq i$, is tried until an initial point X_{k0} is found such that equation $X = X_{k0} + AD^t$ has an integer solution. This can be a very computationally expensive procedure. In contrast, in the PVA, blocks of partitions are explicitly expressed in terms of the vectors. To see which block a given index point $\bar{j} \in Z^n$ belongs to, the computations required are to compute $\Pi\bar{j} (\mathrm{mod}\ \alpha)$ and $\Psi\bar{j}$. Finally, it must be said that despite its disadvantages, the MDM provided the initial inspiration for the ideas of this paper.

B. Smith Normal Form Approach Versus Partitioning Vector Approach

In the Smith normal form approach (abbreviated SNFA), a partitioning matrix U and the displacement vector $\bar{s}$ are used to construct the U-partition. As indicated by Definition 5.2, if an element $s_i, 1 \le i \le m'$, of $\bar{s}$ is equal to one, then $U_i\bar{j} (\mathrm{mod} 1) = 0$ for any arbitrary index point j. This means that all index points $\bar{j} \in J$ are assigned to the same block by U_i. Therefore, when the value of $U_{\bar{j}} (\mathrm{mod}\ \bar{s})$ is computed in Step 2 of Procedure 5.1, the ith row U_i of the partitioning matrix U can be ignored. Because $s_1 | s_2 | \cdots | s_{m'}$, it follows that $s_1 \le \cdots \le s_{m'}$ and only the first $k \le m'$ elements of $\bar{s}$ can possibly be one. If the first k elements of the displacement vector $\bar{s}$ are equal to one, i.e., $s_1 = \cdots = s_k = 1, 0 \le k \le m'$, then the first k rows of matrix U can be removed and only the last $n - k$ rows are needed in Step 2 of Procedure 5.1 to construct the U-partition. When $m' = n$ and $k \ge n - 1$, then only one row (vector) is needed to construct the U-partition as in the PVA. The following theorem provides a sufficient condition such that only the last row of U is needed to construct the U-partition.

Theorem 6.1: Given algorithm (J, D), let $\bar{s}$ be the displacement vector. If there exists a partitioning vector Π such that $disp\Pi = |\det(TD_c)|$ where D_c is the determining matrix of Π and $T = \begin{bmatrix} E_{r_1} \\ \cdots \\ E_{r_{m'}} \end{bmatrix}$ is such that $\mathrm{rank}(TD_c) = m'$, then $s_1 = \cdots = s_{m'-1} = 1$.

Proof: Provided in Appendix.

Theorem 6.1 implies that if $n - m' + 1$ vectors are used to construct the $\alpha\Psi$-partition, then only $n - m' + 1$ rows of matrix U are needed to construct the U-partition. Therefore, it is not true that the SNFA always needs more vectors to construct the U-partition than the PVA to construct the $\alpha\Psi$-partition if $m' > 1$.

Compared with the SNFA, the PVA has the following advantages and disadvantages. First, the SNFA always provides

the maximal pseudo-independent partitions for any uniform dependence algorithms. In contrast, the PVA provides the maximal pseudo-independent partitions only when the uniform dependence algorithm satisfies the condition of Theorem 4.3a. Second, most algorithms to find Smith normal forms have exponential complexity. For the algorithm proposed in [6], although of polynomial complexity, the constant and the exponent are large; in particular, large memory space is needed. When $m = n = m'$, the complexity of the PVA is $O(n^5) + O(n|J|)$ and the complexity of the SNFA is $O(n^{10}) + O(n^2|J|)$. So, generally speaking, the PVA is less computationally expensive than the SNFA. Third, in MIMD systems, one problem is to find an optimal time schedule such that the total execution time plus the total overhead caused by communication is minimized. In this case the PVA is preferred because the partitioning vector Π could also be used to specify a linear schedule [20], [21].

VII. Future Work and Conclusions

Instead of finding maximal independent partitions, it may also be desirable to find maximal dependent partitions such that communication among blocks can be supported by the target machine. For nonpartitionable algorithms (an algorithm is nonpartitionable if its pseudo-partitionability is equal to one), sometimes, it may be desirable to find the maximal partition such that the ratio of communication between a block and other blocks with the cardinality of that block is minimized. The results presented in this paper can be used as a basis for future work dealing with these problems. The main contributions of this paper are two computationally inexpensive methods to identify independent partitions of algorithms with uniform dependencies. The resulting partitions are maximal. These methods can be applied in practice as one of the many analysis procedures used by optimizing compilers to detect and exploit concurrency in serial programs. They may be particularly useful in mapping algorithms into multiprocessor machines where processors are organized in clusters with limited intercluster communication capabilities. In these systems, different clusters can process distinct blocks of a partition without intercluster communication overhead costs. Among others, such multiprocessors include Cedar [9] and Cm^* [5].

Appendix

Before the proofs are presented, some mathematical notations are introduced. These notations are based on [23]. Consider a matrix $A \in \mathbb{R}^{n\times n}$ where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = [\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_n].$$

The cofactors of A are denoted $A_{ij}, i, j = 1, \cdots, n$. The adjugate (or adjoint) matrix of A is denoted A^*. Some facts [23] are also listed here which are used in some of the following proofs.

Fact 1:

$$A^* = \begin{bmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{bmatrix}.$$

Fact 2: $A^{-1} = \frac{A^*}{\det A}$

Lemma 8.1: Let $A = [\bar{a}_1, \cdots, \bar{a}_m] \in IR^{n\times m}$, $\text{rank}(A) = m', \bar{b} = \begin{bmatrix} b_1 \\ \cdots \\ b_n \end{bmatrix} \in sp\{\bar{a}_1, \cdots, \bar{a}_m\}$ and $T \in IR^{m'\times n}$ be such that $\text{rank}(TA) = m'$. Then $\bar{x}$ is a solution of equation $TA\bar{x} = T\bar{b}$ if and only if it is a solution of equation $A\bar{x} = \bar{b}$.

Proof: $\Rightarrow$: Let $M = [A, \bar{b}] = \begin{bmatrix} M_1 \\ \cdots \\ M_n \end{bmatrix}$. Since $\bar{b} \in sp\{\bar{a}_1, \cdots, \bar{a}_m\}$, $\text{rank}(M) = m'$ and $\text{rank}(TM) = m'$. Let $TM = [TA, T\bar{b}] = K = \begin{bmatrix} K_1 \\ \cdots \\ K_{m'} \end{bmatrix}$. Since $K_1, \cdots, K_{m'}$ are linear combinations of $M_1, \cdots, M_n$ and $\text{rank}(K) = \text{rank}(M) = m', sp\{M_1, \cdots, M_n\} = sp\{K_1, \cdots, K_{m'}\}$, i.e., M_i, $i = 1, \cdots, n$, can be expressed as linear combinations of $K_1, \cdots, K_{m'}$. Let

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1m'} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2m'} \\ \cdots & \cdots & \cdots & \cdots \\ \gamma_{n1} & \gamma_{n2} & \cdots & \gamma_{nm'} \end{bmatrix} \begin{bmatrix} K_1 \\ \cdots \\ K_{m'} \end{bmatrix}$$

or $[A\bar{b}] = \Gamma K = \Gamma[TA\ T\bar{b}]$. Then $A = \Gamma TA$ and $\bar{b} = \Gamma T\bar{b}$. Let $\bar{x}$ be a solution of equation $TA\bar{x} = T\bar{b}$, then $\Gamma TA\bar{x} = \Gamma T\bar{b}$, i.e., $A\bar{x} = \bar{b}$. Therefore, $\bar{x}$ is also a solution of $A\bar{x} = b$.

$\Leftarrow$: Let $\bar{x}$ be a solution for equation $A\bar{x} = \bar{b}$. Then $TA\bar{x} = T\bar{b}$ which implies $\bar{x}$ is also a solution of equation $TA\bar{x} = T\bar{b}$. □

The proofs of Lemma 5.1 and Theorem 5.2 are presented first because the proof of Lemma 8.2 becomes much simpler if the results of Theorem 5.2 are used.

Proof of Lemma 5.1:

$\Rightarrow$: Suppose $\bar{j}_1$ and $\bar{j}_2$ are psuedo-connected, then by Definition 2.4, there exists an integer vector $\lambda \in Z^m$ such that $D\bar{\lambda} = \bar{j}_1 - \bar{j}_2$. Let $V \in Z^{m\times m}$ be such that $UDV = S$ is the SNF of matrix D. Then $SV^{-1}\lambda = U(\bar{j}_1 - \bar{j}_2)$. The fact that V is unimodular implies that V^{-1} is also an integer matrix and therefore, $V^{-1}\lambda$ is an integer vector. So, $U(\bar{j}_1 - \bar{j}_2)(\text{mod } \bar{s}) = SV^{-1}\bar{\lambda}(\text{mod } \bar{s}) = \bar{0}$. i.e., $U\bar{j}_1(\text{mod } \bar{s}) = U\bar{j}_2(\text{mod } \bar{s})$.

$\Leftarrow$:: Suppose $U\bar{j}_1(\text{mod } \bar{s}) = U\bar{j}_2(\text{mod } \bar{s})$, i.e., $U(\bar{j}_1 - \bar{j}_2) = [y_1 s_1, y_2 s_2, \cdots, y_{m'} s_{m'}, 0, \cdots, 0]^T$ where y_i, $i = 1, \cdots, m'$, are integers. Let $\bar{y} = [y_1, y_2, \cdots, y_{m'}, 0, \cdots, 0]^T \in Z^m$, then $U(\bar{j}_1 - \bar{j}_2) = S\bar{y}$. This implies that $\bar{j}_1 - \bar{j}_2 = U^{-1}UDV\bar{y}$, or $\bar{j}_1 - \bar{j}_2 = DV\bar{y}$. The fact that V is integral implies that $V\bar{y}$ is integral. So, there exists an integral vector $\bar{\lambda} = V\bar{y}$ such that $\bar{j}_1 - \bar{j}_2 = D\bar{\lambda}$ which means $\bar{j}_1$ and $\bar{j}_2$ are psuedo-connected. □

Proof of Theorem 5.2: Let $\bar{s}$ be the displacement vector.

1) First, it is shown that P_U is a psuedo-independent partition. For any two arbitrary points $\bar{j}_1 \in J_{\bar{y}_i} \in P_U$ and $\bar{j}_2 \in J_{\bar{y}_l} \in P_U$, $\bar{y}_i \neq \bar{y}_l$, by Definition 5.2, $U\bar{j}_1(\text{mod } \bar{s}) = \bar{y}_i$ and $U\bar{j}_2(\text{mod } \bar{s}) = \bar{y}_l$. Because $\bar{y}_i \neq \bar{y}_l$, $U\bar{j}_1(\text{mod } \bar{s}) \neq$

$U\vec{j}_2(\text{mod } \vec{s})$. By Lemma 5.1, $\vec{j}_1$ and $\vec{j}_2$ are not psuedo-connected which implies that P_U is psuedo-independent. Next, it is shown that P_U is the maximal psuedo-independent partition. For any two arbitrary points $\vec{j}_1, \vec{j}_2 \in J_{\bar{y}} \in P_U$, by definition of the U-partition (Definition 5.2), $U\vec{j}_1(\text{mod } \vec{s}) = U\vec{j}_2(\text{mod } \vec{s}) = \bar{y}$. By Lemma 5.1, $\vec{j}_1$ and $\vec{j}_2$ are psuedo-connected. By Definition 2.5, P_U is the maximal psuedo-independent partition.

2) Let $P_U = \{J_{\bar{y}_1}, \cdots, J_{\bar{y}_u}\}$. Consider a block $J_{\bar{y}} \in P_U$ where $\bar{y} = [y_1, \cdots, y_n]^T$. Clearly, $0 \le y_i < s_i$, $i = 1, \cdots, m'$ and $\delta_{il} \le y_k \le \delta_{iu}$, $k = m'+1, \cdots, n$. So there are at most $s_1 \times s_2 \times \cdots \times s_{m'} \times \gamma_{m'+1} \times \cdots \times \gamma_n$ distinct block indexes, i.e., $|P_{\max}| = |P_U| \le \prod_{k=1}^{m'} s_k \prod_{i=m'+1}^{n} \gamma_i$. □

Proof of Theorem 3.1:

1) Suppose $\vec{j}_1$ and $\vec{j}_2$ are pseudo-connected, then there exists a vector $\bar{\lambda} = [\lambda_1 \cdots, \lambda_m]^T \in Z^m$ such that $\vec{j}_1 + D\bar{\lambda} = \vec{j}_2$. Therefore,

$$\Pi\vec{j} + \Pi D\bar{\lambda} = \Pi\vec{j}_2$$

or

$$\Pi\vec{j}_1 + \sum_{i=1}^{m} \lambda_i \Pi\vec{d}_i = \Pi\vec{j}_2.$$

By Definition 3.1, $\gcd(\Pi\vec{d}_1, \cdots, \Pi\vec{d}_m) = \alpha$. This implies that $\Pi\vec{d}_i = \alpha_i\beta_i$ where $\beta_i \in Z$, $i = 1, \cdots, m$. So,

$$\Pi\vec{j}_2 - \Pi\vec{j}_1 = \alpha \sum_{i=1}^{m} \beta_i\lambda_i$$

where $\sum_{i=1}^{m} \lambda_i\beta_i$ is an integer because λ_i and β_i, $i = 1, \cdots, m$, are integers. Then,

$$\Pi(\vec{j}_2 - \vec{j}_1)(\text{mod } \alpha) = 0,$$

i.e., $\Pi\vec{j}_1(\text{mod } \alpha) = \Pi\vec{j}_2 \text{ (mod } \alpha)$. This contradicts to the assumption. So $\vec{j}_1$ and $\vec{j}_2$ are not pseudo-connected.

Consider the α-partition P_α. Since $\Pi\vec{j}(\text{mod } \alpha) = i, \vec{j} \in J_i \in P_\alpha$, $i = 0, \cdots, \alpha - 1$, for any two arbitrary index points $\vec{j} \in J_i, \vec{j}' \in J_l, i \ne l, \Pi\vec{j}(\text{mod } \alpha) \ne \Pi\vec{j}'(\text{mod } \alpha)$ and they are not pseudo-connected. By Definition 2.5, P is a pseudo-independent partition.

2) Suppose that $\vec{j}_1, \vec{j}_2$ are pseudo-connected. Then there exists a vector $\bar{\lambda} \in Z^m$ such that $D\bar{\lambda} = (\vec{j}_1 - \vec{j}_2)$ and, therefore, $\Psi D\bar{\lambda} = \Psi(\vec{j}_1 - \vec{j}_2)$. By Definition 3.3, $\Psi D = \bar{0}$ which implies that $\Psi(\vec{j}_1 - \vec{j}_2) = \bar{0}$, i.e., $\Psi\vec{j}_1 = \Psi\vec{j}_2$. This contradicts the assumption. So, $\vec{j}_1, \vec{j}_2$ are not pseudo-connected. For the Ψ-partition P_Ψ, let $\vec{j}_1 \in J_{\bar{y}_i}$ and $\vec{j}_2 \in J_{\bar{y}_l}, \bar{y}_i \ne \bar{y}_l, J_{\bar{y}_l}, J_{\bar{y}_l} \in P_\Psi$. The fact that $\bar{y}_i \ne \bar{y}_l$ implies that $\Psi\vec{j}_1 \ne \Psi\vec{j}_2$. So, $\vec{j}_1, \vec{j}_2$ are not pseudo-connected. By Definition 2.5, P_Ψ is pseudo-independent.

3) Similarly, let $\vec{j}_1 \in J_{\bar{y}_i} \in P_{\alpha\Psi}$ and $\vec{j}_2 \in J_{\bar{y}_l} \in P_{\alpha\Psi}, \bar{y}_i \ne \bar{y}_l$, where $\bar{y}_i = [y_{0i}, y_{1i}, \cdots, y_{(n-m')i}]^T$ and $\bar{y}_l = [y_{0l}, y_{1l}, \cdots, y_{(n-m')l}]^T$. Since $\bar{y}_i \ne \bar{y}_l$, there exists at least one dimension $t \in \{0, 1, \cdots, n-m'\}$ such that $y_{ti} \ne y_{tl}$. If $t = 0$, then $\Pi\vec{j}_1(\text{mod } \alpha) \ne \Pi\vec{j}_2(\text{mod } \alpha)$ and by 1) of Theorem 3.1, $\vec{j}_1$ and $\vec{j}_2$ are not pseudo-connected. If $1 \le t \le n - m'$, then $\Psi_t\vec{j}_1 \ne \Psi_t\vec{j}_2$ and by 2) of Theorem 3.1, $\vec{j}_1$ and $\vec{j}_2$ are not pseudo-connected. So, by Definition 2.5, $P_{\alpha\Psi}$ is pseudo-independent. □

Proof of Theorem 3.2: If it can be shown that there exists at least on index point $\vec{j} \in J$ such that $\Pi\vec{j}(\text{mod } \alpha) = i$, then $J_i \ne \varnothing$, where $J_i \in P_\alpha, i = 0, \cdots, \alpha - 1$. This implies that the maximal psuedo-independent partition contains at least α blocks. So, $|P_{\max}| \ge \alpha$.

Let $\Pi = [\pi_1\ \pi_2\ \cdots\ \pi_n]$. Since Π is a nonzero vector, it has at least one nonzero component. Without loss of generality, let $\pi_1 \ne 0$. Let $M = t\alpha + i$ (i.e., $M(\text{mod } \alpha) = i$), $M \in Z - \{0\}$, $t \in Z$, and $M \cdot \pi_1 > 0$. Since $\gcd(\pi_1, \cdots, \pi_n) = 1$, by [12] there exists at least one integer solution of the following equation

$$\pi_1\lambda_1 + \cdots + \pi_n\lambda_n = M. \tag{8.1}$$

Let $\bar{z} = \begin{bmatrix} z_1 \\ \cdots \\ z_n \end{bmatrix}$ be such an integer solution of (8.1). If $z_1, \cdots, z_n \ge 0$, then $\bar{z} \in J$ and it has been proven that for any arbitrary integer i, $0 \le i < \alpha$, there exists at least one index point $\bar{z} \in J$ such that $\Pi\bar{z}(\text{mod } \alpha) = M(\text{mod } \alpha) = i$. Now suppose not all $z_1, \cdots, z_n \ge 0$. It is clear that all the solutions of (8.1) take the form

$$\bar{\lambda} = \bar{z} + \begin{bmatrix} -\pi_2 \\ \pi_1 \\ 0 \\ 0 \\ \cdots \\ 0 \end{bmatrix} t_2 + \begin{bmatrix} -\pi_3 \\ 0 \\ \pi_1 \\ 0 \\ \cdots \\ 0 \end{bmatrix} t_3 + \cdots + \begin{bmatrix} -\pi_n \\ 0 \\ 0 \\ \cdots \\ 0 \\ \pi_1 \end{bmatrix} t_n \tag{8.2}$$

where $t_2, t_3, \cdots, t_n$ are constants. This can be verified as follows.

$$\Pi\bar{\lambda} = \Pi\bar{z} + \Pi \begin{bmatrix} -\pi_2 \\ \pi_1 \\ 0 \\ 0 \\ \cdots \\ 0 \end{bmatrix} t_2 + \Pi \begin{bmatrix} -\pi_3 \\ 0 \\ \pi_1 \\ 0 \\ \cdots \\ 0 \end{bmatrix} t_3 + \cdots + \Pi \begin{bmatrix} -\pi_n \\ 0 \\ 0 \\ \cdots \\ 0 \\ \pi_1 \end{bmatrix} t_n$$

$$= M + (-\pi_1\pi_2 + \pi_2\pi_1)t_2 + (-\pi_1\pi_3 + \pi_3\pi_1)t_3 + \cdots + (-\pi_1\pi_n + \pi_n\pi_1)t_n = M.$$

Therefore, $\bar{\lambda}$ is a solution of (8.1). Next, it is shown how a nonnegative integer solution of (8.1) is constructed from (8.2). Let,

$$\bar{\lambda} = \begin{bmatrix} z_1 - \pi_2 t_2 - \pi_3 t_3 - \cdots - \pi_n t_n \\ z_2 + \pi_1 t_2 \\ z_3 + \pi t_3 \\ \cdots \\ z_n + \pi_1 t_n \end{bmatrix} = \begin{bmatrix} \lambda_1 \\ \cdots \\ \lambda_n \end{bmatrix}.$$

If $t_i \ge -\frac{z_i}{\pi_1}$, then $\lambda_i \ge 0, i = 2, \cdots, n$. Let

$$t_i = \left\lceil -\frac{z_i}{\pi_1} \right\rceil = -\frac{z_1}{\pi_1} + \beta_i \text{ where } 0 \le \beta_i < 1 \quad i = 2, \cdots, n. \tag{8.3}$$

Now it is shown that λ_1 is also greater than or equal to zero if t_i, $i = 2, \cdots, n$, are defined by (8.3):

$$\begin{aligned}\lambda_1 &= z_1 - \sum_{i=2}^{n} \pi_i t_i = z_1 - \sum_{i=2}^{n} \left(\beta_i - \frac{z_i}{\Pi_i}\right)\pi_i \\ &= z_1 - \sum_{i=2}^{n} \beta_i \pi_i + \sum_{i=2}^{n} \pi_i \frac{z_i}{\Pi_i} \\ &= z_i - \sum_{i=2}^{n} \beta_i \pi_i + \frac{1}{\pi_1}(M - z_1 \pi_1) \\ &= \frac{M}{\pi_1} - \sum_{i=2}^{n} \beta_i \pi_i.\end{aligned}$$

Notice that $\sum_{i=2}^{n} |\pi_i| > \sum_{i=2}^{n} \beta_i |\pi_i| \geq \sum_{i=2}^{n} \beta_i \pi_i$. If M is selected such that $M = t\alpha + i$ and $\frac{M}{\Pi_1} \geq \sum_{i=2}^{n} |\pi_i|$, then $\lambda_1 = \frac{M}{\pi_1} - \sum_{i=2}^{n} \beta_i \pi_i \geq \frac{M}{\pi_1} - \sum_{i=2}^{n} |\pi_i| \geq 0$. So, there exists an index point $\bar{\lambda} \in J$ such that $\Pi\bar{\lambda} (\mathrm{mod}\ \alpha) = i$. □

Proof of Theorem 4.1:

⇒: Let Π be an equal partitioning vector of (J, D). Then by Definition 3.1, there exists a set of m' linearly independent dependence vectors such that

$$\Pi\bar{d}_{t_1} = \cdots \bar{d}_{t_{m'}} = disp\,\Pi > 0$$

and

$$\Pi\bar{d}_j = a_j disp\Pi,\ a_j \in Z, \quad j = 1, \cdots, m. \tag{8.4}$$

Since $\mathrm{rank}(D) = m'$ and $\bar{d}_{t_1}, \bar{d}_{t_2}, \cdots, \bar{d}_{t_{m'}}$ are linearly independent, $\bar{d}_j$, $1 \leq j \leq m$, can be expressed as a linear combination of $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$, i.e.,

$$\bar{d}_j = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}] \begin{bmatrix} a_{1_j} \\ a_{2_j} \\ \cdots \\ a_{m'_j} \end{bmatrix} \quad 1 \leq j \leq m \tag{8.5}$$

So,

$$\begin{aligned}\Pi\bar{d}_j &= \Pi[\bar{d}_{t_1}, \ldots, \bar{d}_{t_{m'}}] \begin{bmatrix} a_{1_j} \\ a_{2_j} \\ \cdots \\ a_{m'_j} \end{bmatrix} \\ &= disp\Pi[11 \cdots 1] \begin{bmatrix} a_{1_j} \\ a_{2_j} \\ \cdots \\ a_{m'_j} \end{bmatrix} \\ &= disp\Pi \sum_{i=1}^{m'} a_{ij}.\end{aligned}$$

By (8.4), $\Pi\bar{d}_j = a_j disp\Pi$, where $a_j \in Z$, $1 \leq j \leq m$. So $\sum_{i=1}^{m'} a_{ij} = a_j$, $1 \leq j \leq m$, are integers.

⇐: Consider the following two systems of equations:

$$\begin{cases} \Pi(\bar{d}_{t_1} - \bar{d}_{t_2}) = 0 \\ \Pi(\bar{d}_{t_1} - \bar{d}_{t_3}) = 0 \\ \cdots \\ \Pi(\bar{d}_{t_1} - \bar{d}_{t_{m'}}) = 0 \end{cases} \tag{8.6}$$

and

$$\begin{cases} \Pi\bar{d}_{t_1} = 0 \\ \Pi\bar{d}_{t_2} = 0 \\ \cdots \\ \Pi\bar{d}_{t_{m'}} = 0. \end{cases} \tag{8.7}$$

Let N_1 be the solution space of (8.6) and N_2 be the solution space of (8.7). If Π is a solution of (8.7), then it is a solution of (8.6). Thus $N_2 \subseteq N_1$. The dimension of N_1 is $n - m' + 1$ and the dimension of N_2 is $n - m'$. This implies that $N_2 \subset N_1$. So, there exists at least one solution Π' of (8.6) such that $\Pi' d_{t_j} \neq 0$, $1 \leq j \leq m'$. Let $\Pi \in Z^{1 \times n}$ be such a solution of (8.6) that $\Pi\bar{d}_{t_j} = a > 0$, $j = 1 \cdots, m'$, and the greatest common divisor of the n components of Π is equal to one. Then

$$\Pi\bar{d}_j = \Pi[\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}] \begin{bmatrix} a_{1_j} \\ a_{2_j} \\ \cdots \\ a_{m'j} \end{bmatrix} = a \sum_{i=1}^{m'} a_{ij}, j = 1, \cdots, m.$$

Since $\sum_{i=1}^{m'} a_{ij}$, $j = 1, \cdots, m$, are integers, the following holds.

$$\Pi\bar{d}_j (\mathrm{mod}\ a) = 0,\ j = 1, \cdots, m \text{ and } disp\Pi = a > 0.$$

By Definition 3.1, Π is an equal partitioning vector of (J, D). □

Proof of Corollary 4.1:

1) Let us first prove that if algorithm (J, D) satisfies the first condition, then it has an equal partitioning vector. Consider the following two systems of equations:

$$\begin{cases} \Pi(\bar{d}_1 - \bar{d}_2) = 0 \\ \Pi(\bar{d}_1 - \bar{d}_3) = 0 \\ \cdots \\ \Pi(\bar{d}_1 - \bar{d}_m) = 0 \end{cases} \tag{8.8}$$

and

$$\begin{cases} \Pi\bar{d}_1 = 0 \\ \Pi\bar{d}_2 = 0 \\ \cdots \\ \Pi\bar{d}_m = 0. \end{cases}$$

Using reasoning similar to that used in the second part of the proof of Theorem 4.1, it can be shown that there exists at least one solution Π' of (8.8) such that $\Pi'\bar{d}_j \neq 0$, $1 \leq j \leq m$. Let $\Pi \in Z^{1 \times n}$ be such a solution of (8.8) that $\Pi\bar{d}_j = disp\,\Pi > 0$, $j = 1, \cdots, m$, and the greatest common divisor of the n components of Π is equal to one. Then $\Pi\bar{d}_j = disp\,\Pi > 0$, $j = 1, \cdots, m$. Therefore, Π, is an equal partitioning vector.

2) By the assumption of condition 2, there exists a set of m' linearly independent dependence vectors $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$ such that all dependence vectors can be expressed as an integer linear combination of $\bar{d}_{t_1}, \cdots, \bar{d}_{t_{m'}}$, i.e., $\bar{d}_j = \sum_{i=1}^{m'} a_{ij}\bar{d}_{t_i}$, $j = 1, \cdots, m$, where a_{ij}, $i = 1, \cdots, m'$ and $j = 1, \cdots, m$, are integer constants. Clearly, $\sum_{i=1}^{m'} a_{ij}$, $j = 1, \cdots, m$, are integers. According to Theorem 4.1, (J, D) has an equal partitioning vector. □

Lemma 8.2: Let $m' = n, \Pi$ be a partitioning vector of algorithm (J, D) determined by $\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}, D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_n}]$ and $\bar{j}_1, \bar{j}_2 \in J$. If $|\det D_c| = disp\ \Pi$ and $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod } disp\ \Pi) = 0$, then $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected.

Proof: Consider algorithm (J, D_c). Its algorithm coefficient α' is equal to $|\det D_c|$ because $\alpha' = \gcd(\Pi\bar{d}_{t_1}, \cdots, \Pi\bar{d}_{t_n}) = \gcd(disp\Pi, \cdots, disp\Pi) = disp\Pi = |\det D_c|$. Let $\bar{s}' = [s'_1, \cdots, s'_n]^T$ be the displacement vector of algorithm (J, D_c). By Theorem 5.1, $\prod_{i=1}^{n} s'_i = |\det D_c|$. By Theorem 5.2, there are at most $|\det D_c|$ blocks in the maximal pseudo-partition of algorithm (J, D_c) and by Corollary 3.2 $|P'_\alpha| = \alpha' = |\det D_c|$. So, the α'-partition is the maximal pseudo-independent partition. Because $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod } disp\ \Pi) = 0, \bar{j}_1$ and $\bar{j}_2$ are in the some block of the α-partition. Therefore, $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected. □

Proof of Theorem 4.3:

1) By Theorem 3.1, $\bar{j}_1, \bar{j}_2$ are pseudo-connected only if $\Pi\bar{j}_1(\text{mod } \alpha) = \Pi\bar{j}_2(\text{mod } \alpha)$. Let us prove that $\bar{j}_1, \bar{j}_2$ are pseudo-connected if $\Pi\bar{j}_1(\text{mod } \alpha) = \Pi\bar{j}_2(\text{mod } \alpha)$. Because $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod } \alpha) = 0$, it follows $\Pi(\bar{j}_1 - \bar{j}_2) = \gamma_1 disp\Pi + \gamma_2\alpha, 0 \le \gamma_2\alpha < disp\Pi$ and $\gamma_1, \gamma_2 \in Z$. If $\gamma_2 = 0$, by Lemma 8.2, $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected. Suppose $\gamma_2 \neq 0$. Because $\gcd(\Pi\bar{d}_1, \cdots, \Pi\bar{d}_m) = \alpha$. By [12], there exists at least one integer solution of the following equation:

$$\lambda_1\Pi\bar{d}_1 + \ldots + \lambda_m\Pi\bar{d}_m = \alpha. \tag{8.9}$$

Let $\bar{\lambda} = [\lambda_1, \cdots, \lambda_m]^T$ be an integer solution of (8.9). Then $\gamma_2\bar{\lambda}$ is an integer solution of the following equation:

$$\gamma_2\lambda_1\Pi\bar{d}_1 + \cdots + \gamma_2\lambda_m\Pi\bar{d}_m = \gamma_2\alpha. \tag{8.9a}$$

Let $\bar{j} = \bar{j}_2 + \gamma_2 D\bar{\lambda}$. Clearly, $\bar{j}$ and $\bar{j}_2$ are pseudo-connected and by (8.9a)

$$\Pi(\bar{j} - \bar{j}_2) = \gamma_2\lambda_1\Pi\bar{d}_1 + \cdots + \gamma_2\lambda_m\Pi\bar{d}_m = \gamma_2\alpha.$$

Therefore, $\Pi(\bar{j}_1 - \bar{j}) = \Pi\bar{j}_1 - \Pi\bar{j} + \Pi\bar{j}_2 - \Pi\bar{j}_2 = \Pi(\bar{j}_1 - \bar{j}_2) - \Pi(\bar{j} - \bar{j}_2) = \gamma_1 disp\Pi + \gamma_2\alpha - \gamma_2\alpha$. So, $\Pi(\bar{j}_1 - \bar{j})(\text{mod } disp\Pi) = 0$. By Lemma 8.2, $\bar{j}_1$ and $\bar{j}$ are pseudo-connected. Since $\bar{j}_1, \bar{j}$ and $\bar{j}, \bar{j}_2$ are pseudo-connected, respectively, $\bar{j}_1$ and $\bar{j}_2$ are also pseudo-connected. Therefore, if $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod } \alpha) = 0$, then $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected.

2) Consider the α-partition P_α, since $\Pi\bar{j}_1(\text{mod } \alpha) = \Pi\bar{j}_2(\text{mod } \alpha)$ where $\bar{j}_1, \bar{j}_2 \in J_i \in P_\alpha$ are arbitrary index points, $0 \le i < \alpha$, by the results in 1) of Theorem 4.3, $\bar{j}_1, \bar{j}_2$ are pseudo-connected. By Definition 2.5, P_α is the maximal pseudo-independent partition, i.e., $P_\alpha = P_{max}$ and $|P_{max}| = |P_\alpha| = \alpha$. □

Lemma 8.3: Let $J_{\bar{y}} \in P_\Psi$, then $J_{\bar{y}} = \{\bar{j} : \bar{j} \in J, \bar{j} = \bar{p} + D\bar{x}, \bar{x} \in \mathbb{R}^m, \Psi\bar{p} = \bar{y}\}$.

Proof: Let S denote $\{\bar{j} : \bar{j} \in J, \bar{j} = \bar{p} + D\bar{x}, \bar{x} \in \mathbb{R}^m, \Psi\bar{p} = \bar{y}\}$. If $\bar{j} \in J_{\bar{y}}$, then $\Psi\bar{j} = \bar{y}$ and $\Psi(\bar{j} - \bar{p}) = \bar{0}$. By the *Fundamental Theorem of Linear Algebra* [23, pp. 15, 87], $\bar{j} - \bar{p}$ belongs to $sp\{\bar{d}_1, \cdots, \bar{d}_m\}$ because $\Psi D = 0$. This implies that there exists an $\bar{x} \in \mathbb{R}^m$ such that $\bar{j} - \bar{p} = D\bar{x}$, i.e., $\bar{j} = \bar{p} + D\bar{x}$. So, $\bar{j} \in S$ and $J_{\bar{y}} \subseteq S$. Now, let $\bar{j} \in S$, then $\bar{j} = \bar{p} + D\bar{x}$ and $\Psi\bar{j} = \Psi\bar{p} + \Psi D\bar{x} = \Psi\bar{p} = \bar{y}$. So $\bar{j} \in J_{\bar{y}}$ and $J_{\bar{y}} = S$. □

Lemma 8.4: Let $\bar{d}_{t_1}, \cdots, \bar{d}_{t_m}$ be linearly independent, $D_c = [\bar{d}_{t_1}, \cdots, \bar{d}_{t_m}], T \in Z^{m' \times n}$ be such that rank $(TD_c) = m', J_{\bar{y}} \in P_\Psi$ and $L_{\bar{y}} = \{T\bar{j} : \bar{j} \in J_{\bar{y}}\}$. Then, 1) the mapping $T : J_{\bar{y}} \to L_{\bar{y}}, T(\bar{j}) = T\bar{j}, \bar{j} \in J_{\bar{y}}$ is bijective and 2) $T\bar{j}_1, T\bar{j}_2 \in L_{\bar{y}}$ are pseudo-connected in algorithm $(L_{\bar{y}}, TD)$ if and only if $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ are pseudo-connected in algorithm $(J_{\bar{y}}, D$, i.e., $\bar{j}_1 - \bar{j}_2 = D\bar{\lambda}$ if and only if $T(\bar{j}_1 - \bar{j}_2) = TD\bar{\lambda}, \bar{\lambda} \in Z^m$.

Proof:

1) Consider the mapping $T : J_{\bar{y}} \to L_{\bar{y}}, T(\bar{j}) = T\bar{j}, \bar{j} \in J_{\bar{y}}$. Since $L_{\bar{y}} = \{T\bar{j} : \bar{j} \in J_{\bar{y}}\}$, T is surjective. By Lemma 8.3, $J_{\bar{y}} = \{\bar{j} : \bar{j} \in J, \bar{j} = \bar{p} + D\bar{x}, \bar{x} \in IR^m, \Psi\bar{p} = \bar{y}\}$. Since $\bar{d}_{t_1}, \cdots, \bar{d}_{t_m}$ are linearly independent and rank$(D) = m'$, each dependence vector can be written as a linear combination of $\bar{d}_{t_1}, \cdots, \bar{d}_{t_m}$, i.e., $D = D_c\Lambda$ where $\Lambda \in \mathbb{R}^{m' \times m}$. So, $J_{\bar{y}}$ can be rewritten as $J_{\bar{y}} = \{\bar{j} : \bar{j} = \bar{p} + D_c\bar{z}, \bar{z} = \Lambda\bar{x}, \bar{j} \in J, \bar{x} \in \mathbb{R}^{m'}, \Psi\bar{p} = \bar{y}\}$. Let $\bar{j}_1, \bar{j}_2 \in J_y$ and $\bar{j}_1 = \bar{p} + D_c\bar{z}_1, \bar{j}_2 = \bar{P} + D_c\bar{z}_2$, then $T(\bar{j}_1 - \bar{j}_2) = TD_c(\bar{z}_1 - \bar{z}_2) = 0$ if and only if $\bar{z}_1 = \bar{z}_2$, or equivalently, $\bar{j}_1 = \bar{j}_2$, since rank$(TD) = m'$. In summary, it has been shown that $T\bar{j}_1 = T\bar{j}_2$ if and only if $\bar{j}_1 = \bar{j}_2$. So T is injective which implies T is bijective.

2) $\Rightarrow$: If $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ are psuedo-connected, then there is a vector $\bar{\lambda} \in Z^m$ such that $\bar{j}_1 = \bar{j}_2 + D\lambda$. So, $T\bar{j}_1 = T\bar{j}_2 + TD\bar{\lambda}$ and $T\bar{j}_1, T\bar{j}_2$ are pseudo-connected.

($<=$) : If $T\bar{j}_1, T\bar{j}_2$ are pseudo-connected, then there exists a vector $\bar{\lambda} \in Z^m$ such that $T(\bar{j}_1 - \bar{j}_2) = TD\bar{\lambda}$. Index points $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ implies that $\bar{j}_1 - \bar{j}_2 \in sp\{\bar{d}_1, \cdots, \bar{d}_m\}$. By Lemma 8.1, $\bar{\lambda}$ is also a solution of equation $D\bar{\lambda} = (\bar{j}_1 - \bar{j}_2)$ which implies that $\bar{j}_1$ and $\bar{j}_2$ are pseudo-connected. □

Proof of Theorem 4.3a:

1) $\Rightarrow$: see Theorem 3.1.

$\Leftarrow$: Since $\Psi\bar{j}_1 = \Psi\bar{j}_2, \bar{j}_1$ and $\bar{j}_2$ belong to the same block of the Ψ-partition, i.e., $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}} \in P_\Psi$, where $\bar{y} = \Psi\bar{j}_1$. Let $L_{\bar{y}} = \{T\bar{j} : \bar{j} \in J_{\bar{y}}\}$ and $\Delta = TD$. Consider the algorithm $(L_{\bar{y}}, \Delta)$, let Γ be the partitioning vector determined by $T\bar{d}_{t_1}, \cdots, T\bar{d}_{t_m}$ and $\alpha' = \gcd(\Gamma T\bar{d}_1, \cdots, \Gamma T\bar{d}_m)$ be the algorithm coefficient for algorithm $(L_{\bar{y}}, \Delta)$. By Theorem 4.2, $\Gamma = \beta' 1(TD_c)^{-1}$ and $disp\ \Gamma = \beta'$, where $\beta' \in N^+$ is such that the greatest common divisor of the m' components of Γ is equal to one and $\Gamma \in Z^{1 \times m'}$. Now, consider the row vector $\Pi = (1/\beta'')\Gamma T = \beta 1(TD_c)^{-1}T$ where $\beta = (\beta'/\beta''), \beta \in N^+$ is such that the greatest common divisor of the n components of Π is equal to unity and $\Pi \in Z^{1 \times n}$. By theorem 4.2, Π is a partitioning vector determined by vectors $\bar{d}_1, \cdots, \bar{d}_m$ for algorithm (J, D). Therefore, the algorithm coefficient for algorithm (J, D) is $\alpha = \gcd(\Pi\bar{d}_1, \cdots, \Pi\bar{d}_m) = \gcd((1/\beta'')\Gamma T\bar{d}_1, \cdots, (1/\beta'')\Gamma T\bar{d}_m) = \alpha'/\beta''$.

By assumption $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod}\alpha) = 0$, i.e., $\Pi(\bar{j}_1 - \bar{j}_2) = \lambda\alpha$ where $\lambda \in Z$. Because $\alpha = \alpha'/\beta''$, $\Pi\bar{j}_1 = (1/\beta'')\Gamma T\bar{j}_1$ and $\Pi\bar{j}_2 = (1/\beta'')\Gamma T\bar{j}_2$, $\Pi(\bar{j}_1 - \bar{j}_2) = (1/\beta'')\Gamma(T\bar{j}_1 - T\bar{j}_2) = \lambda\alpha'/\beta''$, or $\Gamma(T\bar{j}_1 - T\bar{j}_2) = \lambda\alpha'$ which means $\Gamma(T\bar{j}_1 - T\bar{j}_2)(\text{mod}\alpha') = 0$. That is, if $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod}\alpha) = 0$, then $\Gamma(T\bar{j}_1 - T\bar{j}_2)(\text{mod } \alpha') = 0$. Notice that the dimension of the index points of algorithm $(L_{\bar{y}}, \Delta)$ is m' and the rank of its dependence matrix Δ is m'. According to Theorem 4.3 1), for any two arbitrary index points $T\bar{j}_1, T\bar{j}_2 \in L_{\bar{y}}$, if $\Gamma(T\bar{j}_1 - T\bar{j}_2)(\text{mod } \alpha') = 0$, then $T\bar{j}_1, T\bar{j}_2$ are pseudo-connected. According to Lemma 8.4, $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}}$ are also pseudo-connected in algorithm $(J_{\bar{y}}, D)$. In summary, it has been proven that for any two arbitrary index points $\bar{j}_1, \bar{j}_2 \in J$, if $\Pi(\bar{j}_1 - \bar{j}_2)(\text{mod}\alpha) = 0$ and $\Psi\bar{j}_1 = \Psi\bar{j}_2$, then $\bar{j}_1, \bar{j}_2$ are pseudo-connected.

2) Consider any two arbitrary points $\bar{j}_1, \bar{j}_2 \in J_{\bar{y}_i} \in P_{\alpha\Psi}$, $i = 1, \cdots, v$. Since $\Pi(\bar{j}_1 - \bar{j}_2) (\text{mod } \alpha) = 0$ and $\Psi(\bar{j}_1 - \bar{j}_2) = \bar{0}$, by the results in Theorem 4.3a 1), they are pseudo-connected. By Definition 2.5, $P_{\alpha\Psi}$ is the maximal pseudo-independent partition.

3) Let $P_\alpha^{\bar{y}}$ denote the α-partition for algorithm $(L_{\bar{y}}, \Delta)$. First, $|P_\alpha^{\bar{y}}| \leq \alpha$ because there are at most α nonempty blocks in $P_\alpha^{\bar{y}}$. Second, for P_Ψ, clearly, there are at most $\prod_{i=1}^{n-m'}(\gamma_i + 1)$ nonempty blocks, where $\gamma_i = \max\{\Psi_i(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}, i = 1, \cdots, n - m'$. So, $|P_\Psi| \leq \prod_{i=1}^{n-m'}(\gamma_i + 1)$. The fact that $|P_\alpha^{\bar{y}}| \leq \alpha$ implies that $|P_{\max}| \leq \max\{|P_\alpha^{\bar{y}}| : J_{\bar{y}} \in P_\Psi\}|P_\Psi| = \alpha|P_\Psi|$. By Theorem 3.1 and Corollary 3.2, P_α is a pseudo-independent partition of (J, D) and $|P_\alpha| = \alpha$. So $\alpha \leq |P_{\max}| \leq \alpha|P_\Psi|$. □

Proof of Theorem 6.1: Without loss of generality, let $\det(TD_c) > 0$. Because $disp\Pi = \det(TD_c)$, by Theorem 4.2, $\Pi = disp\Pi\bar{1}(TD_c)^{-1}T = \det(TD_c)\bar{1}(TD_c)^{-1}T$. By fact 2, $\Pi = \bar{1}(TD_c)^*T$. Let $\Gamma = [\gamma_1, \cdots, \gamma_{m'}] = \bar{1}(TD_c)^*$, then $\gcd(\gamma_1, \cdots, \gamma_{m'}) = \gcd(\pi_1, \cdots, \pi_n) = 1$. Let $(TD_c)_{ij}$, $i, j = 1, \cdots, m'$ be the cofactors of matrix (TD_c). Then by Fact 1, $\Gamma = [\sum_{j=1}^{m'}(TD_c)_{1j}, \cdots, \sum_{j=1}^{m'}(TD_c)_{m'j}]$, i.e., $\gamma_i = \sum_{j=1}^{m'}(TD_c)_{ij}, i = 1, \cdots, m'$. So, $\gcd(\sum_{j=1}^{m'}(TD_c)_{1j}, \cdots, \sum_{j=1}^{m'}(TD_c)_{m'j}) = \gcd(\gamma_1, \cdots, \gamma_{m'}) = 1$ which means $\gcd((TD_c)_{ij}, i, j = 1, \cdots, m') = 1$. This implies that the greatest common divisor of all subdeterminants of order $m' - 1$ is equal to 1. By Theorem 5.1, $\prod_{i=1}^{m'-1} s_i = 1$. Therefore, $s_1 = \cdots = s_{m'-1} = 1$. □

List of Symbols

D: dependence matrix with n rows and m columns; see Definition 2.1 4).
D_c: determining matrix; see Definition 3.1.
$\bar{d}_i$: dependence (column) vector with n components; see Definition 2.1 4).
$\det(A)$: determinant of matrix A.
$disp\Pi$: a positive integer; see Definition 3.1.
E: the set of edges of the algorithm dependence graph; see Definition 2.2.
E_i: row vector with n components whose entries are zero except that the ith entry is one.
I: identity matrix.
$\mathbb{R}$: set of real numbers.
J: index set; see Definition 2.1 1).
J_i: a block of a partition; see Definitions 2.3, 2.5 and 3.2.
$J_{\bar{y}}$: a block of a partition; see Definitions 3.4, 3.5 and 5.2.
$\bar{j}$: index point (column vector); see Definition 2.1 1).
m: number of dependence vectors in D; see Definition 2.1 4).
m': the rank of the dependence matrix D; see Definition 2.1 4).
N: set of nonnegative integers.
N^+: set of positive integers.
n: number of components of index points in J; see Definition 2.1 1).
P: a pseudo-independent partition; see Definition 2.5.
$P_{\max}$: maximal pseudo-independent partition; see Definition 2.5.
P_α: α-partition; see Definition 3.2.
P_Ψ: Ψ-partition; see Definition 3.4.
$P_{\alpha\Psi}$: $\alpha\Psi$-partition; see Definition 3.5.
P_U: U-partition; see Definition 5.2.
ρ: independent partition; see Definition 2.3.
$\text{rank}(A)$: rank of matrix A.
S: the Smith normal form for dependence matrix D; see Theorem 5.1.
$\bar{s}$: displacement vector (column); see Definition 5.1.
$sp\{\bar{v}_1, \cdots, \bar{v}_k\}$: the vector space spanned by vectors $\bar{v}_1, \cdots, \bar{v}_k$.
SNF: abbreviation of Smith Normal Form.
U: left multiplier of the Smith normal form; see Theorem 5.1.
V: right multiplier of the Smith normal form; see Theorem 5.1.
$\bar{x}$: a real column vector.
$\bar{y}$: index vector of blocks of a partition; see Definitions 3.4, 3.5, and 5.2.
Z: set of integers.
α: algorithm coefficient (positive integer); see Definition 3.1.
μ: number of blocks in the U-partition; see Definition 5.2.
v: number of blocks in the $\alpha\Psi$-partition; see Definition 3.5.
Π: partitioning vector (row); see Definition 3.1.
Ψ: separating matrix see Definition 3.3.

Ψ_i: separating vector (row); see Definition 3.3.
ψ: number of blocks in the Ψ-partition; see Definition 3.4.
$\emptyset$: empty set.
$\bar{1}$: a column or row vector whose entries are all 1.
$\bar{0}$: a column or row vector whose entries are all 0.
$|P|$: cardinality of set P.
$|c|$: the absolute value of scalar c.
$A - B$: set $\{x : x \in A, x \notin B\}$, where A and B are sets.
$a|b$: a divides b, $a, b \in Z$.
$b \pmod a$: modulo operation, i.e., $b \pmod a = d$ iff $a|(b - d)$ where $0 \leq d < a$.

ACKNOWLEDGMENT

The authors thank S. H. Zak, M. O'Keefe, and V. Van Dongen for their discussions on Section V. Also, J.-K. Peir provided valuable insights on the MDM and explained how it could be used to partition the algorithm of Example 5.2 in [22].

REFERENCES

[1] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle, "Time and parallel processor bounds for FORTRAN-like loops," *IEEE Trans. Comput.*, vol. C-28, pp. 660–670, Sept. 1979.
[2] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 836–844.
[3] J. A. B. Fortes, "Algorithm transformations for parallel processing and VLSI architecture design," Ph.D. dissertation, Dep. Elec. Eng.–Syst., Univ. of Southern California, Dec. 1983.
[4] D. D. Gajski and J.-K. Peir, "Essential issues in multiprocessor systems," *IEEE Comput. Mag.*, vol. 18, pp. 9–27, June 1985.
[5] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.
[6] R. Kannan and A. Bachem, "Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix," *SIAM J. Comput.*, vol. 8, no. 4, pp. 499–507, Nov. 1979.
[7] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, July 1967.
[8] D. C. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. Kruskal, "The effects of program restructuring, algorithm changes, and architecture choice on program performance," in *Proc. 1984 Int. Conf. Parallel Processing*, pp. 129–138.
[9] D. Kuck, E. Davidson, D. Lawrie, and A. H. Sameh, "Parallel supercomputing today and the Cedar approach," *Science*, vol. 231, pp. 967–974, Feb. 28, 1986.
[10] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. C-35, pp. 1–12, Jan. 1986.
[11] D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, vol. 71, pp. 113–120, Jan. 1983.
[12] L. Mordell, *Diophantine Equations*. New York: Academic, 1969, p. 30.
[13] D. A. Padua, "Multiprocessors: Discussion of theoretical and practical problems," Ph.D. dissertation, Rep. UIUCDCS-R-79-990, Univ. of Illinois at Urbana-Champaign, Urbana, IL, Nov. 1979.
[14] D. A. Padua, D. J. Kuck, and D. L. Lawrie, "High speed multiprocessor and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, pp. 763–776, Sept. 1980.
[15] J.-K. Peir and R. Cytron, "Minimum distance: A method for partitioning recurrences for multiprocessors," in *Proc. 1987 Int. Conf. Parallel Processing*, pp. 217–225.
[16] J.-K. Peir and D. D. Gajski, "CAMP: A programming aid for multiprocessors," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 475–482.
[17] J.-K. Peir, "Program partitioning and synchronization on multiprocessor systems," Ph.D dissertation, Rep. UIUCDCS-R-86-1259, Dep. Comput. Sci., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Mar. 1986.
[18] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "Execution of parallel loops on parallel processor systems," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 519–527.
[19] A. Schrijver, *Theory of Linear and Integer Programming*. New York: Wiley, 1986.
[20] W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," in *Proc. Int. Conf. Systolic Arrays*, May 1988, pp. 393–402.
[21] ——, "Partitioning of uniform dependency algorithms for parallel execution on MIMD/systolic systems," Tech. Rep. TR-EE 88-18, School of Electrical Engineering, Purdue Univ., W. Lafayette, IN 47907, Apr. 1988.
[22] ——, "Independent partitioning of algorithms with uniform dependencies," in *Proc. 1988 Int. Conf. Parallel Processing, Vol 2 Software*, pp. 26–33.
[23] G. Strang, *Linear Algebra and its Applications*, second ed. New York: Academic, 1980.
[24] O. Veblen and P. Franklin, "On matrices whose elements are integers," *Ann. of Mathematics*, vol. 23 (1921–2), pp. 1–15.
[25] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D dissertation, Rep. UIUCDCS-R-82-1105, Univ. of Illinois at Urbana-Champaign, Urbana, IL, 1982

Weijia Shang (S'88–M'90) received the B.S. degree in computer engineering and science from Changsha Institute of Technology in 1982 and the M.S. and Ph.D. degrees in electrical engineering from Purdue University, West Lafayette, IN, in 1984 and 1990, respectively.

She is currently an Assistant Professor in the Center for Advanced Computer Studies, the University of Southwestern Louisiana, Lafayette. Her research interests include parallel processing, computer architecture, algorithm transformation, processor array programming, special purpose VLSI bit-level processor array design, and optimizing compiler technique.

Dr. Shang is a member of the Association for Computing Machinery.

Jose A. B. Fortes (S'80–M'83) received the Licenciatura em Engenharia Electrotecnica from the Universidade de Angola in 1978, the M.S. degree in electrical engineering from Colorado State University, Fort Collins, in 1981, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1984.

In 1984, he joined the faculty of the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he currently is an Associate Professor. From July 1989 through July 1990 he served at the National Science Foundation as program director for microelectronics systems architecture. His research interests are in the areas of parallel processing, fault-tolerant computing, and VLSI computer architecture on which he co-authored over 50 technical papers. His research has been funded by the Office of Naval Research, AT&T Foundation, General Electric, and the National Science Foundation.

Dr. Fortes is a member of the IEEE Professional Society. He is on the Editorial Boards of the *Journal of Parallel and Distributed Computing* and the *Journal of VLSI Signal Processing*.

REFERENCE NO. 5

Rau, D., Fortes, J. A. B. and Siegel, H. J., "Destination Tag Routing Techniques Based on a State Model for the IADM Network," *IEEE Transactions on Computers*, Volume 41, Number 3, March 1992, pp. 274-286.

Note - This paper proposes a simple destination-tag routing mechanism for a class of networks. It also shows how the message routing scheme supports the rerouting of messages around faulty nodes and links. These results are of use in fault-tolerant processor arrays where a multistage network is used to connect processors.

Destination Tag Routing Techniques Based on a State Model for the IADM Network

Darwen Rau, Jose A. B. Fortes, *Member, IEEE*, and Howard Jay Siegel, *Fellow, IEEE*

Abstract—**A "state model" is proposed for solving the problem of routing and rerouting messages in the Inverse Augmented Data Manipulator (IADM) network. Using this model, necessary and sufficient conditions for the reroutability of messages are established, and then destination tag schemes are derived. These schemes are simpler, more efficient, and require less complex hardware than previously proposed routing schemes. Two destination tag schemes are proposed. For one of the schemes, rerouting is totally transparent to the sender of the message and any blocked link of a given type can be avoided. Compared with previous works that deal with the same type of blockage, the time × space complexity is reduced from $O(\log N)$ to $O(1)$. For the other scheme, rerouting is possible for any type of link blockage. A universal rerouting algorithm is constructed based on the second scheme, which finds a blockage-free path for any combination of multiple blockages if there exists such a path, and indicates absence of such a path if there exists none. In addition, the state model is used to derive constructively a lower bound on the number of subgraphs which are isomorphic to the Indirect Binary n-Cube network in the IADM network. This knowledge can be used to characterize properties of the IADM networks and for permutation routing in the IADM networks.**

Index Terms—**Cube network, data manipulator network, destination-tag routing, fault tolerance, interconnection network, multiprocessor, parallel processing, state model.**

I. Introduction

THIS paper discusses novel and efficient techniques for routing and rerouting messages in the Inverse Augmented Data Manipulator (IADM) network [9]. These results are based on a new approach, the "state model," which characterizes and correlates the topologies of the IADM and Indirect Binary n-Cube networks, and leads to efficient exploitation of the redundancy available in the IADM network.

Considerable research has been dedicated to the design of multistage interconnection networks for multiprocessor systems. The class of data manipulator networks, introduced in [3], includes, among others, the Augmented Data Manipulator (ADM) network [17], the IADM network [9], and the Gamma network [13], [14]. The IADM network and the ADM network differ only in that the input side of one of them corresponds to the output side of the other and vice versa. The Gamma and the IADM networks are topologically equivalent; however, they use switches of different types. Each 3 × 3 crossbar switch used in the Gamma network can connect simultaneously all three inputs to all three outputs whereas each switch used in the IADM network can connect only one of its three inputs to one or more of its three outputs. The main interest of this paper is the study of the IADM network; both the one-to-one and permutation routings are considered. The schemes proposed for routing and rerouting messages in the IADM network are also applicable to the Gamma network.

Perhaps the most popular class of multistage networks is the multistage *cube-type* networks such as the Indirect Binary n-Cube [15], Omega [6], Baseline [20], Generalized Cube [18], STARAN flip [2], and a special case of SW-Banyan [4] networks. Among the main advantages of these networks are their very efficient destination tag routing schemes, partitionability, $O(N \log_2 N)$ cost, and ability to pass useful permutations [16]. Some results of this paper are based on characteristics of the Indirect Binary n-Cube network (hereon referred to as the *ICube network*). Since the cube-type networks mentioned above are all topologically equivalent [16], [17], [20], [21], the results in this paper are also relevant to any of them.

The ICube network is composed of $n = \log N$ stages labeled from 0 to $n-1$. Each stage consists of N connection links and $N/2$ interchange (switches) boxes. The structure of the network is such that two input links of an interchange box differ only in the ith bit of their labels; the upper links have a "0" in the ith bit and the lower links have a "1." Fig. 1 illustrates an ICube network of size $N = 8$ and two possible states of an interchange box, "straight" and "exchange." Since this paper considers only one-to-one and permutation routing, broadcast states are not shown.

The *IADM* network is composed of n stages labeled from 0 to $n-1$. Each stage consists of $3N$ connection links and N switching elements. An extra column of switches is appended at the end of the last stage as the output switches and is referred to as stage n. Each switch j at stage i has three output links to switches $(j - 2^i) \bmod N$, j, and $(j + 2^i) \bmod N$ of the succeeding stage. Each switch selects one of its input links and connects it to one or more output links. Fig. 2 illustrates an IADM network of size $N = 8$.

In a multistage interconnection network, the path connecting the source of a message to its destination is determined by a routing scheme that specifies the switching state of each

Manuscript received October 15, 1987; revised August 12, 1991. This work was supported in part by the National Science Foundation under Grant DCI-8419745, by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contract 00014-88-k-0723, and by the Supercomputing Research Center under Contract MDA904-85-C-5027.

D. Rau was with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907. He is now with AT&T Bell Laboratories, Naperville, IL 60566.

J. A. B. Fortes and H. J. Siegel are with the Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9105560.

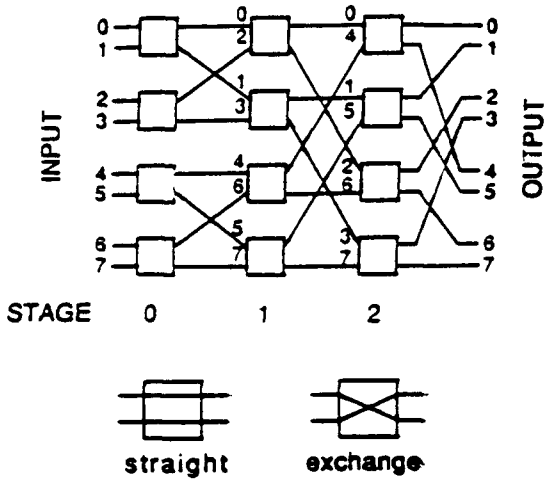


Fig. 1. The Indirect Binary n-Cube (ICube) network for $N = 8$ (according to the first graph model); two possible states for each box are shown (i.e., straight and exchange).

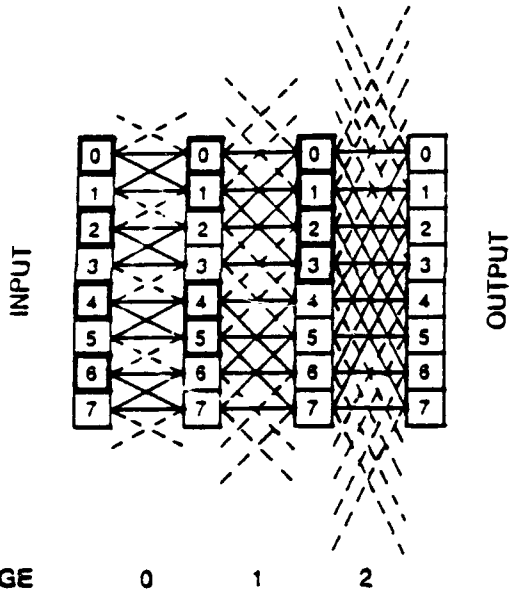


Fig. 2. The IADM network for $N = 8$ (according to the first graph model); *even*$_i$ and *odd*$_i$ switches, $0 \le i \le 2$, are enclosed with bold and regular edges, respectively. The solid edges (links) show the ICube subgraph.

switch in the path. Routing schemes are considerably simpler for the cube-type networks than for the data manipulator-type networks. In cube-type networks, the interchange box at stage i needs to examine the ith bit of the binary representation of the destination address of an incoming message. If the ith bit is 0, then the upper output of the box is taken. If the ith bit is 1, the lower output of the box is taken. These schemes are known as *destination tag routing schemes* [6] and are extremely efficient and simple to implement. Unlike cube-type networks, in the IADM and other data manipulator-type networks there are several paths between any source s and destination d ($s \neq d$) and each switching element has at least three switching states. Previously proposed routing schemes [9], [10], [13] for the IADM network can be thought of as *distance tag schemes*; that is, they require calculation of the distance from source to destination in order to generate routing and rerouting tags. The rerouting schemes in these works are basically finding an alternate representation, which specifies an alternate routing path, for the distance.

McMillen and Siegel [9] proposed three dynamic rerouting techniques for the IADM network for avoiding faulty or blocked $\pm 2^i$ (nonstraight) links. The first and the second schemes require that switches be capable of performing two's complement and $\pm 2^i$ addition operations, respectively. The third scheme requires one extra tag bit which is dynamically updated as the message propagates toward the destination. In [10], the work of [9] was expanded, and a single-stage look-ahead scheme was proposed to avoid certain types of straight link faults. This improved scheme also requires two's complement operations.

Parker and Raghavendra [13] used redundant number representation and proposed an algorithm capable of finding all routing paths, which, effectively, are the redundant number representations for the distance between the source and the destination. Because of the complexity of the algorithm, the cost of computation is prohibitively large so that it is infeasible to implement the algorithm in order to achieve dynamic routing [19]. In addition, although the algorithm can generate all routing tags for any distance, there is no specific work on rerouting schemes in [13], [14].

Lee and Lee [7] proposed signed bit difference tag and destination tag local control algorithms for the ADM and IADM networks that require no computation for the distance between the source and the destination. But their local control algorithms can only find one routing path for each source and destination pair. If the need for rerouting arises, they still resort to the distance tag schemes to find alternate paths.

Past research has shown interesting relationships between data manipulator and cube-type networks. For example, because it is possible to embed the Generalized Cube network in the ADM network [1], [17], the set of interconnections implementable by the ADM network is a superset of that of the Generalized Cube network. This fact and the existence of multiple paths between any source s and destination d ($s \neq d$) in the ADM network suggests that the ADM network can be thought of as a fault-tolerant Generalized Cube network. Analogously, the IADM network can be regarded as a fault-tolerant ICube network.[1] Since the permutations realizable by cube-type networks are well studied, the identification of possible embeddings of the ICube network in the IADM network can help characterize the permutation capabilities of this network. A contribution to the precise understanding of these notions is made in this paper; it consists of the identification of a large number of distinct subgraphs of the IADM network that are isomorphic to the ICube network.

Section II of this paper introduces a state model to describe and correlate topologies of the ICube network and the IADM network. Necessary and sufficient conditions to perform rerouting in the IADM network are derived in Section III. In Section IV, two routing and rerouting schemes are proposed

[1] While topologically equivalent, the ICube and Generalized Cube I/O ports are addressed so that their interrelationship is the same as that of the IADM and ADM network, i.e., the input and output sides are interchanged.

based on the theory developed in Section III, together with a discussion of their merits and implementation considerations. A universal rerouting algorithm is proposed in Section V, which can deal with any combination of multiple link blockages. A class of subgraphs in the IADM network that are isomorphic to the ICube network are identified in Section VI, and it is shown how to reconfigure the IADM network under certain link faults to pass the cube-admissible permutations. Finally, Section VII summarizes the results presented in this paper.

II. State Model Description for the ICube and IADM Networks

Multistage networks can be modeled as graphs by treating interchange boxes (also called switching elements) and links of the network as nodes and edges of the graph, respectively. Another equivalent graph model [1], [8] results if interchange boxes are associated with edges, and links with nodes. Both models are exemplified in Figs. 1 and 3 for the ICube network. The IADM network is shown in Fig. 2 according to the first model. The design of switches based on both models is discussed in [11]. Clearly, the ICube network in Fig. 3 can be regarded as being a subgraph of the IADM network in Fig. 2. Henceforth, the second model is always assumed when referring to the ICube network (i.e., Fig. 2) and the first model is assumed when dealing with the IADM network.

With respect to these graph models, the nodes and the edges of the graph refer to the switches and the links of the networks, respectively. The number of switches at each stage of a network is denoted N and $n = \log_2 N$ refers to the number of stages. The switches of each stage are labeled from 0 to $N-1$ from the top to the bottom. Any integer j has a binary representation $j_0 j_1 \cdots j_{n-1}$, where j_{n-1} is the most significant bit and n denotes the number of bits. The notation $j_{r/s}$ means the bits of j starting at j_r and ending at j_s, where $r \le s$. Bit $\overline{j_i}$ is 1's complement of bit j_i. Throughout this paper, j and $j+a$, where a is some constant, are reserved to represent labels of switches. Also modulo N arithmetic is assumed, e.g., $j+a$ implies $(j+a) \bmod N$. The notation $j \in S_i$ is used to indicate that a switch j belongs to stage i and $(j' \in S_i, j'' \in S_{i+1})$ is used to represent a link at stage i joining $j' \in S_i$ and $j'' \in S_{i+1}$. A sequence of switches of contiguous stages $(j' \in S_i, j'' \in S_{i+1}, \cdots j''' \in S_{i+k})$ is used to represent a path from $j' \in S_i$ to $j''' \in S_{i+k}$.

Notation and terminology required for the characterization of network topologies and destination tag routing schemes are introduced next. A switch j of stage i is an $even_i$ switch if $j_i = 0$ and an odd_i switch if $j_i = 1$. Fig. 2 identifies $even_i$ and odd_i switches at different stages of the IADM network of size $N = 8$. Let t_i represent a tag bit and define the functions ΔC_i and $\Delta \overline{C}_i$ that represent connection links at stage i as

$$\Delta C_i(j, t_i) = \begin{cases} 0 & \text{if } j \text{ is an } even_i \text{ switch and } t_i = 0 \\ 0 & \text{if } j \text{ is an } odd_i \text{ switch and } t_i = 1 \\ -2^i & \text{if } j \text{ is an } odd_i \text{ switch and } t_i = 0 \\ +2^i & \text{if } j \text{ is an } even_i \text{ switch and } t_i = 1 \end{cases}$$

$$\Delta \overline{C}_i(j, t_i) = -\Delta C_i(j, t_i).$$

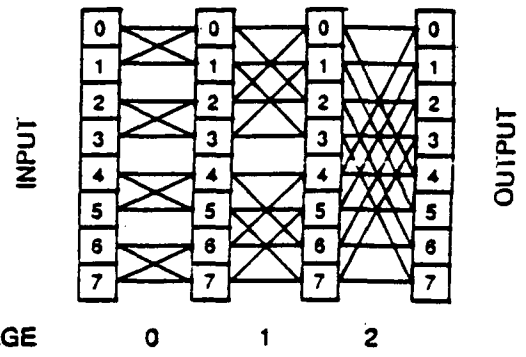


Fig. 3. The Indirect Binary n-Cube (ICube) network for $N = 8$ (according to the second graph model).

Also, define the functions $C_i(j, t_i) = j + \Delta C_i(j, t_i)$ and $\overline{C}_i(j, t_i) = j + \Delta \overline{C}_i(j, t_i)$. These definitions imply the following lemma of fundamental importance to the results of this paper.

Lemma 2.1:

$$C_i(j, t_i) = j_{0/i-1} t_i j_{i+1/n-1}$$
$$\overline{C}_i(j, t_i) = j_{0/i-1} t_i q_{i+1/n-1}$$

for some value of $q_{i+1/n-1}$ which depends on j and t_i.

Proof: If j is an $even_i$ switch and $t_i = 0$, then $C_i(j, t_i) = \overline{C}_i(j, t_i) = j$. If j is an odd_i switch and $t_i = 1$, then $C_i(j, t_i) = \overline{C}_i(j, t_i) = j$. If j is an odd_i switch, $t_i = 0$, then $C_i(j, t_i)$ results from subtracting 1 from j_i. Since j is an odd_i switch, $j_i = 1$, no borrow is generated and all remaining bits of j are unchanged; however, $\overline{C}_i(j, t_i)$ adds 1 to j_i, changing the ith bit to 0 and altering some of the bits in positions $i+1, \cdots n-1$ due to carry propagation. Similar reasoning applies when j is an $even_i$ switch and $t_i = 1$. □

The notation and terminology just introduced can now be used to describe the networks of interest in this paper. The following description for a network in terms of ΔC_i, $\Delta \overline{C}_i$, C_i and $\overline{C}_i$ is called the *network state model*.

The ICube network is composed of n stages labeled from 0 to $n-1$. Each stage consists of $2N$ links and N switches. An extra column of switches is appended at the end of the last stage as the output switches (Fig. 3) and is denoted S_n. A switch $j \in S_i$ is connected to switches $C_i(j, t_i) \in S_{i+1}$, for $0 \le i \le n-1$, $0 \le j \le N-1$, and $t_i = 0$ or $t_i = 1$. When using destination tags, switch $j \in S_i$ routes a message to switch $C_i(j, d_i) \in S_{i+1}$ where d_i is the ith bit of the address of the message destination.

The IADM network is composed of n stages labeled from 0 to $n-1$. Each stage consists of a column of N switches and $3N$ connection links. An extra column of switches is appended at the end of the last stage as the output switches and is denoted S_n. A switch $j \in S_i$ is connected to switches $C_i(j, t_i) \in S_{i+1}$ and $\overline{C}_i(j, t_i) \in S_{i+1}$ for $0 \le i \le n-1$, $0 \le j \le N-1$, and $t_i = 0$ or $t_i = 1$. In other words, three links connect a switch $j \in S_i$ to the switches $(j - 2^i)$, j and $(j + 2^i)$ at stage $i+1$. Sometimes $+2^i$ and -2^i are used to represent links $(j \in S_i, (j + 2^i) \in S_{i+1})$ and $(j \in S_i, (j - 2^i) \in S_{i+1})$, respectively. The term *a straight link* refers to link

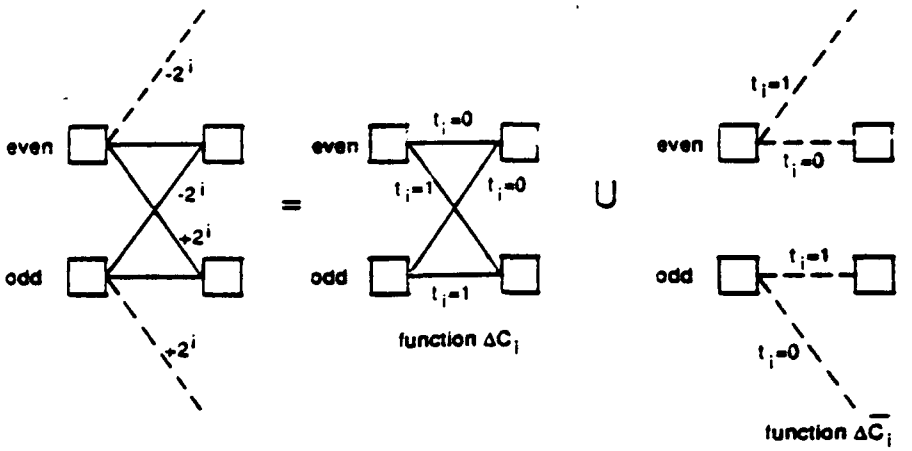


Fig. 4. The connection links of stage i of the ICube network can be described by the function ΔC_i. The connection links of stage i of the IADM network can be described by the union of the functions ΔC_i and $\Delta\overline{C}_i$.

$(j \in S_i, j \in S_{i+1})$ and the term *a nonstraight link* refers to links $\pm 2^i$.

According to the model, two types of switches, $even_i$ and odd_i are required in the IADM and ICube networks. Fig. 4 illustrates the connection links of a pair of $even_i$ and odd_i switches for an ICube and IADM network of size $N = 8$. The ΔC_i function describes the ICube connections. For the IADM network, the connection links can be described by the union of the functions ΔC_i and $\Delta\overline{C}_i$. In practice, $even_i$ and odd_i switches can be identical and easily programmed (at power-up or system configuration time) to behave differently.

There are two possible routing behaviors (or states) for each switch in an IADM network. A switch is said to be in *state* C if the routing is decided in accordance with the function $C_i(j, t_i)$ and it is in the *state* $\overline{C}$ if the function $\overline{C}_i(j, t_i)$ applies. On the whole, the link on which a message is routed depends on whether the switch is an $even_i$ or odd_i switch, in state C or $\overline{C}$, and the value of tag bit t_i. Also the term *state of the network* is used to denote collectively the states of all switches in the network.

The notion of switch state is only conceptual; it can be implemented by designing the switches with actual logic states as well as by using tags with n added bits specifying the states of the switches on the routing path. In Section IV, these and other aspects of the actual implementation of the proposed schemes are discussed in detail.

III. Theory Behind the State-Based Destination Tag Routing Schemes

Based on the framework developed in Section II, routing problems in the IADM network are now examined. It is clear that when every switch in the IADM network is in the state C, the IADM network behaves like an ICube network and, therefore, the destination address $d_{0/n-1}$ can be used as a routing tag, i.e., $t_i = d_i$. More generally, the following theorem can be proven.

Theorem 3.1: Let $d \triangleq d_{0/n-1}$ be the destination in the IADM network to which a message is to be sent. Then $t = d_{0/n-1}$ is the unique destination routing tag to the destination d regardless of state of the IADM network.

Proof: Consider an arbitrary tag $f_{0/n-1}$ and assume that the IADM network is in an arbitrary state. Let $t_{0/n-1} = f_{0/n-1}$. Then each switch will route the incoming message to either $C_i(j, f_i)$ or $\overline{C}_{i(j,f_i)}$. From Lemma 2.1, it can be reasoned by induction that, at stage i, $(C_i(f, f_i))_{0/i} = (\overline{C}_i(j, f_i))_{0/i} = f_{0/i}$; at the last stage, $C_{n-1}(j, f_{n-1}) = \overline{C}_{n-1}(j, f_{n-1}) = f_{0/n-1}$. Thus, the address of the destination of the message is the same as the routing tag. This proves both the validity and the uniqueness of $d_{0/n-1}$ as a routing tag. □

It is implicit in the reasoning underlying Theorem 3.1 that any link on a given path results from the appropriate choice of the state of the corresponding switch, i.e., the use of "link" $\Delta C_i(j, t_i)$ results from setting $j \in S_i$ to state C and use of "link" $\Delta\overline{C}_i(j, t_i)$ results from setting $j \in S_i$ to state $\overline{C}$. Thus, given a path to the destination d, there is at least one network state for which the use of d as the destination tag results in the routing of a message through that path.

The implication of Theorem 3.1 is that the use of a state model for the IADM network reduces the problem of finding alternate routing paths to that of controlling the states of the switches in the network. Capitalizing on this idea, the following theorems show how alternate routing paths can be found in order to evade blockages in the network. A *straight link blockage* occurs if a straight link on the routing path is faulty or busy. A *nonstraight link blockage* is defined analogously. The third type of blockage, called *double nonstraight link blockage*, occurs if both nonstraight output links of a switch in the routing path are faulty or busy. A *switch blockage* occurs if the switch itself is busy or faulty. A switch blockage has the same effect as blocking all of the switch's input links and can be transformed into a link blockages problem accordingly. The discussion on rerouting in this paper is concerned only with link blockages.

Theorem 3.2: In the IADM network, a change of the state of switch $j \in S_i$ results in a different routing path to a destination d if and only if a nonstraight output link of j is used on the

original routing path to d. Moreover, the other nonstraight output link of j is used on the new path.

Proof: Changing the state of j implies that the "link" $\Delta C_i(j, t_i)$ is used instead of $\Delta \overline{C}_i(j, t_i)$ or vice versa. However, if $\Delta C_i(j, t_i) = 0$ then $\Delta \overline{C}_i(j, t_i) = 0$ (i.e., both use a straight link) and vice versa. □

With regard to the rerouting schemes proposed in this paper, the implications of Theorem 3.2 are twofold. First, the "if" part of the theorem implies that dynamic rerouting for a nonstraight link blockage can be achieved by changing the state of the switch whose output is the nonstraight link, which is equivalent to rerouting the message through the oppositely signed nonstraight link connected to the same switch. Thus, the same subset of destinations is reachable from the two switches whose input links are the two oppositely signed nonstraight links. Second, the "only if" part of the theorem implies that dynamic rerouting for a straight link blockage is impossible. This is true in general since every routing path in the IADM network can be the result of setting the network to some state. Moreover, if a path from stage i' to stage i'' consists of all straight links connecting $j \in S_i$ and $j \in S_{i+1}$, $i' \leq i \leq i''$, then there exist no alternate routing paths from $j \in S_{i'}$ to $j \in S_{i''}$ for otherwise there would exist an alternate routing path branching from $j \in S_{i'}$ and ending at the destination. The only resort, if any at all, to bypass the straight link blockage is to backtrack to a switch connected to a nonstraight link on the routing path at some preceding stage and to reroute from that switch. It remains to show that an alternate routing path always exists, provided that such a nonstraight link exists. In fact, the existence of an alternate routing path partly results from Theorem 3.2, as stated in the next theorem. Fig. 5 illustrates the situation in Theorem 3.3 for which a proof is provided in [22].

Theorem 3.3: Consider a routing path in the IADM network to a destination d that contains a blocked straight link at stage i. There exists at least one network state which results in an alternate routing path that avoids the same straight link blockage at stage i if and only if the original routing path to d contains a nonstraight link at stage $i-k$ for some k, $i \geq k > 0$.

Previous work [7], [9], [13] implies only the "if" part of the theorem, i.e., the possibility of using nonstraight link of opposite sign in order to reroute a message in the case of a nonstraight link failure. However the "only if" part of the theorem also implies that, in addition, it is not possible to devise a new rerouting scheme capable of avoiding a backtracking (or look-ahead) mechanism in order to deal with straight link blockages.

From Theorem 3.2, (for a given source/destination pair) if the straight output link of a switch is on some routing path, both nonstraight output links of the switch cannot be used for routing; if one of the nonstraight output links of a switch is on some routing path, the other nonstraight link of the switch is also on another routing path and the straight link of the switch cannot be used for routing. So, for a given switch, the output link blockages that affect paths from a given source to a given destination can only be 1) a nonstraight link blockage, 2) a straight link blockage, or 3) a double nonstraight

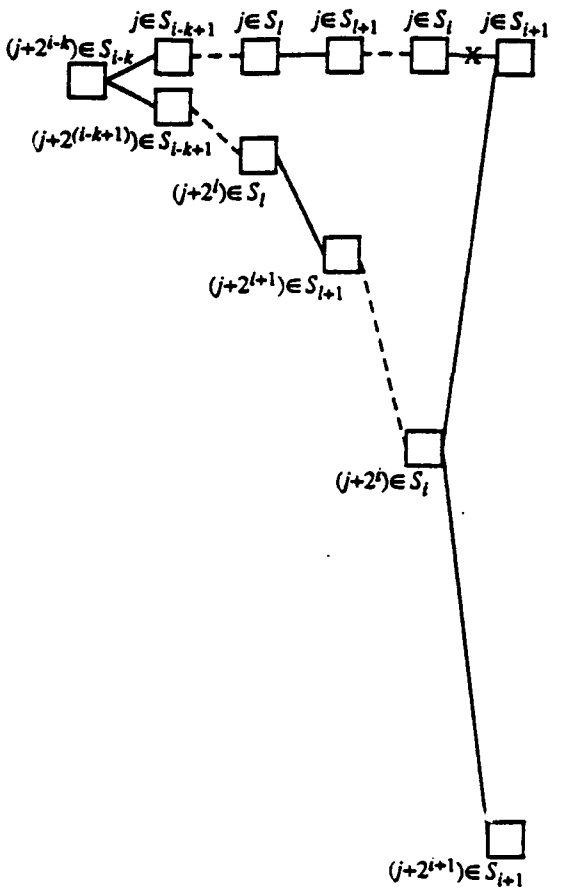


Fig. 5. Rerouting for a straight link blockage in $(j \in S_i, j \in S_{i+1})$. Path $((j + 2^{i-k}) \in S_{i-k}, j \in S_{i-k}, \cdots, j \in S_{i+1})$ is a segment of the original path: $((i + 2^{i-k} \in S_{i-k}, (j + 2^{i-k+1}) \in S_{i-k+1}, \cdots, j \in S_{i+1})$ and $((j + 2^{i-k}) \in S_{i-k}, (j + 2^{i-k+1}) \in S_{i-k+1}, \cdots, (j + 2^i) \in S_i, (j + 2^{i+1}) \in S_{i+1})$ are the rerouting paths for it.

link blockage.[2] Theorem 3.2 can be used to avoid case 1) a nonstraight link blockage and Theorem 3.3, case 2) a straight link blockage. If case 2) occurs, then Theorem 3.2 cannot be used to find a rerouting path. A backtracking scheme proposed later in Corollary 4.2 based on Theorem 3.3 can be adapted to overcome this type of blockage. The adapted backtracking scheme is based on Theorem 3.4, which is illustrated in Fig. 6. The proof of Theorem 3.4 is provided in [22].

Theorem 3.4: Consider a routing path in the IADM network to a destination d that contains a switch at stage i whose both nonstraight output links are blocked. There exists at least one network state which results in an alternate routing path that avoids the same blocked nonstraight links at stage i if and only if the original routing path to d contains a nonstraight link at stage $i - k$ for some k, $i \geq k > 0$.

IV. State-Based Routing and Rerouting Schemes

In this section, routing and rerouting schemes are discussed based on the theory developed in Section III. As mentioned

[2] Physically it is possible to have any combination of blockages of the output links of a given switch. However, the possible routing paths for a given source/destination pair can be affected by either a straight link blockage or a double nonstraight link blockage in a given switch but never both types of blockage.

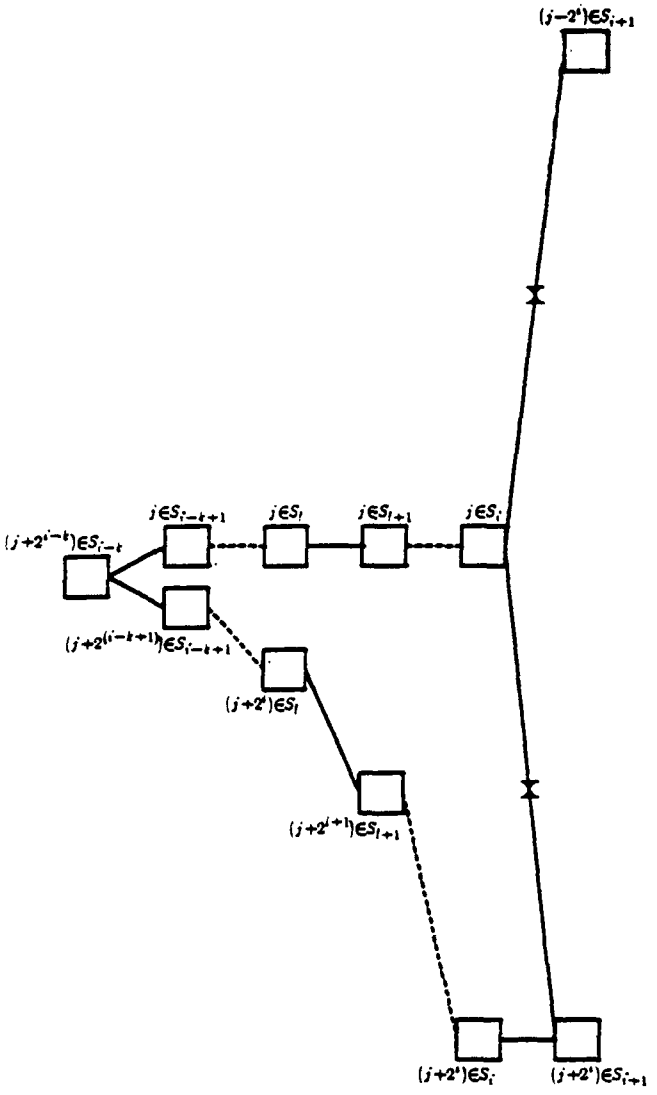


Fig. 6. Rerouting for a double nonstraight links blockage in $(j \in S_i, (j-2^i) \in S_{i+1})$ and $(j \in S_i, (j+2^i) \in S_{i+1})$. Path $((j+2^{i-k}) \in S_{i-k}, (j+2^{i-k+1}) \in S_{i-k+1}, \cdots, (j+2^i) \in S_i, (j+2^i) \in S_{i+1})$ is a rerouting path for both paths $((j+2^{k-1}) \in S_{i-k}, j \in S_{i-k+1}, \cdots, j \in S_{i+1}, (j-2^i) \in S_{i+1})$ and $((j+2^{k-1}) \in S_{i-k}, j \in S_{i-k+1}, \cdots, j \in S_{i+1}, (j+2^i) \in S_{i+1})$.

earlier, the novelty of the ideas in this paper lies in the state model of the routing behavior of each switch. In previously proposed approaches, routing is determined solely by tag bits. According to the state model, the switching action of each network element is conceptually determined by its relative position (i.e., an $even_i$ or odd_i switch), its state (i.e., C or $\overline{C}$) and a destination tag bit (i.e., 0 or 1) (Fig. 4). This conceptual separation of routing information makes it possible to devise the simple routing schemes described in this section.

In the first scheme, each switch is initially set up to behave as an odd_i or $even_i$ switch. In addition, each switch can dynamically be set to one of the logical states C or $\overline{C}$. In other words, this scheme corresponds to a direct implementation of the conceptual view of switch states. Destination tags are used and, according to Theorem 3.1, the state of the network is transparent to the sender of the message since it only affects the path of the message and not its destination. Consequently, rerouting is also transparent in the sense that it results from a change in the network state. In practice, the implementation can be such that, for instance state C (or $\overline{C}$) is used as the default state for each switch in the IADM network and the switch regarcs the other nonstraight link as a spare link for rerouting; if a nonstraight blockage is detected, then the switch changes state to $\overline{C}$ (or C) so that the spare link is used instead. This scheme is called the *Self-Repairing State-Based Destination Tag (SSDT)* scheme.

Rerouting is useful not only when one nonstraight link in a switch is faulty or busy, but also if both nonstraight links are busy. For example, when considering a packet switching environment, rerouting may be desirable as a means of balancing the message load throughout the network. The scheme proposed here is well suited for this purpose. Assume that each nonstraight link has an associated buffer (queue). When both nonstraight links are busy due to message traffic congestion, a switch can choose which nonstraight buffer to assign a message to (i.e., which state to associate with that queued message), based on the number of messages present in the buffers in order to evenly distribute the message load to the nonstraight links.

The proposed SSDT scheme has the advantages that it uses simple n-bit destination tags and is capable of rerouting messages when blockages occur in nonstraight links. In addition, rerouting of a message is transparent to its sender since the path of the message is determined by the state of the network. For a given destination tag, the routing behavior of each switch on a possible path is determined by the state of the switch, i.e., the SSDT scheme is fully distributed and rerouting is done dynamically. Each switch requires a negligible amount of extra hardware for the detection of blocked links and the representation of two possible states.

The second scheme is called the *Two-bit State-Based Destination Tag (TSDT)* scheme and it uses $2n$-bit routing tags, which specify both the destination of the message and the states of switches on the corresponding path. The TSDT scheme has the advantage that rerouting is possible when blockages occur for straight as well as nonstraight links.

As with the first scheme, the TSDT scheme assumes that each switch is appropriately initialized to behave as an odd_i or $even_i$ switch. Each "digit" of the routing tag is represented by two bits b_{n+i} and b_i, called the *state bit* and the *destination bit*, respectively. For this scheme, the state of a switch of stage i is specified by b_{n+i}: if $b_{n+i} = 0$, the switch is in state C and if $b_{n+i} = 1$, the switch is in state $\overline{C}$. For all i, $0 \le i \le n-1$, $b_i = d_i$. In general, if j is an $even_i$ switch, $b_i b_{n+i} = 00$ and $b_i b_{n+i} = 01$ direct the message through a straight link, $b_i b_{n+i} = 10$ through link $+2^i$ and $b_i b_{n+i} = 11$ through link -2^i, if j is an odd_i switch $b_i b_{n+i} = 10$ and $b_i b_{n+i} = 11$ direct the message through a straight link, $b_i b_{n+i} = 01$ through link $+2^i$ and $b_i b_{n+i} = 00$ through link -2^i. In general, given a switch, the destination bit specifies use of a straight link or a nonstraight link while the state bit determines the choice of the positive or the negative link (if the chosen link is a nonstraight link). Since state information is carried by the routing tag, switches are not required to determine and remember their own states, i.e., the design of the switches does not need to

implement the logic states C and $\overline{C}$.

From Theorem 3.2, a nonstraight link blockage at stage i can be bypassed conveniently by complementing the ith state bit while the destination bits remain unchanged. For convenience of reference, this is restated in terms of the TSDT scheme as Corollary 4.1 below.

Corollary 4.1: Let $b_{n/2n-1}$ and $b'_{n/2n-1}$ be the state bits of the routing tag and the rerouting tag, respectively, for the IADM network. In order to bypass a nonstraight link blockage at stage i, state bit b_{n+i} needs to be change to $\overline{b}_{n+i}$. That is, $b'_{n/2n-1} = b_{n/n+i-1}\overline{b}_{n+i}b_{n+i+1/2n-1}$. □

Fig. 7 illustrates an example of routing from $s = 1$ to $d = 0$ in an IADM network of size $N = 8$. Let $b_{0/5} = 000000$ be the routing tag and $b'_{0/5}$ and $b''_{0/5}$ denote the rerouting tags. The original tag $b_{0/5} = 000000$ specifies the path $(1 \in S_0, 0 \in S_1, 0 \in S_2, 0 \in S_3)$. If $(1 \in S_0, 0 \in S_1)$ is blocked, the rerouting tag $b'_{0/5} = 000100$ is obtained by complementing b_3, and link $(1 \in S_0, 2 \in S_1)$ is used for rerouting. This tag specifies the path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$. If $(2 \in S_1, 0 \in S_2)$ is also blocked, the rerouting tag $b''_{0/5} = 000110$ results from complementing b'_4, and link $(2 \in S_1, 4 \in S_2)$ is used for rerouting. This tag specifies the path $(1 \in S_0, 2 \in S_1, 4 \in S_2, 0 \in S_3)$.

As discussed in Section III, a straight link blockage and a double nonstraight link blockage cannot be overcome easily; implementing a backtracking (or look-ahead) mechanism is a must in order to evade these types of blockages. Since all links in the routing path from stage $i - k + 1$ to stage i consist of only straight links, backtracking of at least k stages is required to find the switch from which an alternate routing path branches. That is, at least k state bits need to be considered for change. Due to the similarity between Theorems 3.3 and 3.4, the TSDT schemes for finding the rerouting paths from Theorems 3.3 and 3.4 are exactly the same, which is stated as Corollary 4.2 (see [22] for the proof).

Corollary 4.2: Let $b_{n/2n-1}$ and $b'_{n/2n-1}$ be the state bits of the routing tag and the rerouting tag, respectively, for a source/destination pair in the IADM network. Let $i - k$ be the largest stage number for $i \geq k > 0$ such that a switch at stage $i - k$ is connected to a nonstraight link on the routing path. In order to bypass a straight link blockage or a double nonstraight link blockage at stage i, only state bits $b_{n+(i-k)/n+i-1}$ need to be changed: a) $b'_{n/n+(i-1)} = b_{n/n+(i-k)-1}\overline{d}_{i-k/i-1}$ if the nonstraight link at stage $i-k$ of the original path is link -2^{i-k}, and b) $b'_{n/n+(i-1)} = b_{n/n+(i-k)-1}d_{i-k/i-1}$ if the nonstraight link at stage $i - k$ of the original path is link $+2^{i-k}$. The state bits $b'_{n+i/2n-1}$ have arbitrary values in both cases.

The example in Fig. 7 can be used to illustrate the TSDT scheme for 1) a straight link blockage and 2) a double nonstraight link blockage. 1) Again the tag $b_{0/5} = 000000$ specifies a path $(1 \in S_0, 0 \in S_1, 0 \in S_2, 0 \in S_3)$. If the straight link $(0 \in S_1, 0 \in S_2)$ is blocked, the rerouting tag can be 000110 which specifies path $(1 \in S_0, 2 \in S_1, 4 \in S_2, 0 \in S_3)$ by having $b'_{3+0}b'_{3+1}b'_{3+2} = \overline{d}_0\overline{d}_1b_{3+2} = 110$. Since state bits $b'_{3+1}b'_{3+2}$ can be arbitrary, 000100, for example, is also a valid rerouting tag; it specifies path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$. 2) Let the tag $b_{0/5} = 000110$ specify a path

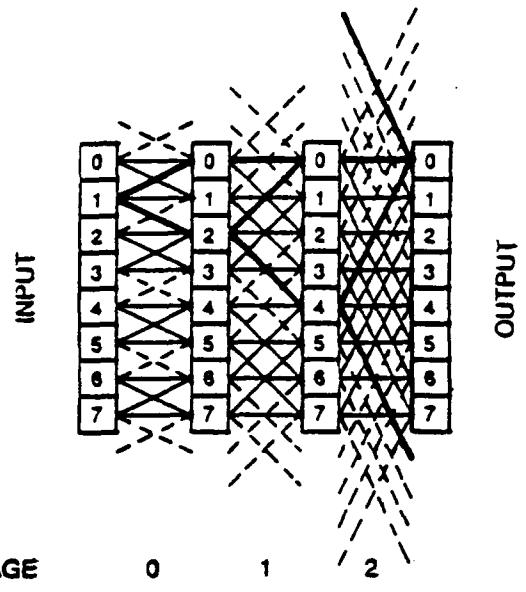


Fig. 7. All routing paths from $1 \in S_0$ to $0 \in S_3$ in an IADM network of size $N = 8$.

$(1 \in S_0, 2 \in S_1, 4 \in S_2, 0 \in S_3)$. If both nonstraight output links of $4 \in S_2$ are blocked, the rerouting tag $b'_{0/5}$ can be 000100 which specifies path $(1 \in S_0, 2 \in S_1, 0 \in S_2, 0 \in S_3)$ by having $b'_{3+0}b'_{3+1}b'_{3+2} = b_{3+0}d_1d_2$. Since state bits b'_{3+2} can be arbitrary, 000101 is also a valid rerouting tag which also specifies the same path.

The rerouting path computed from Corollary 4.2 is blockage-free from stage 0 to stage i. While the rerouting path is different from the original routing path from stage $i - k$ to stage i, the routing path from stage 0 to $i - k - 1$ remains the same. This results from the fact that backtracking always proceeds backward along the original path until it stops at stage $i - k$, and the rerouting path only changes course from stage $i - k$ onwards. Although state bits $b_{n+i/2n-1}$ remain unchanged, the routing path from stage i to $n - 1$ may still be altered due to the changes from stage $i - k$ to i. For example, in Fig. 5, the switch on the original routing path at stage $i + 1$ is $j \in S_{i+1}$ whereas the switch on the rerouting path at stage $i + 1$ may be $(j + 2^{i+1}) \in S_{i+1}$, which may further induce changes at higher-order stages.

In the TSDT scheme, the tag can be computed by the message sender which is assumed to know the location of faulty links and switches in the network. Thus, rerouting is transparent to the switches in the sense that the tag computed by the sender of the message simply avoids the usage of faulty links and switches. Therefore, switches do not require any extra hardware for rerouting purposes. It is assumed that there exists a fault detection and location mechanism and a process that maintains a list of faulty switches and links. Changes to the list are broadcasted by this process to the senders (the exact broadcast method depends on system implementation). For better performance, these lists should be in fast access memory (e.g., cache). An alternative is to implement dynamic rerouting for the TSDT scheme. Since backtracking is indispensable for avoiding a straight link blockage, it is required that each switch can detect the inaccessibility of any output port (connected to

a switch at the next stage) and signal the presence of the blockage back to the switches of previous stages [10], [12]. Whether rerouting is done by the sender or dynamically is an implementation decision which depends on how many stages of backtracking are allowed. When the sender computes the tag, it must be able to identify and track the switches and links on the corresponding routing and rerouting paths (the next paragraphs explain how this is done). If any of the switches or links in the path is known to the sender as being faulty, then the sender computes another tag by changing the state bits as described in Section V.

Locating the switches on the routing path is straightforward. For a given source s and a destination d, the initial routing path can be specified by setting state bits $b_{n/2n-1} = 0_{n/2n-1}$ (a string of n 0's), equivalent to setting every switch in the IADM network to state C. Then every switch on the original path has label $d_{0/i-1}s_{i/n-1} \in S_i$, $0 \le i \le n-1$, since now the IADM network functions like an ICube network [6], [15].

To find the switches on the rerouting path, let $j \in S_i$ be the switch whose output link is blocked. First consider the case where the blocked link is a nonstraight link. It may be a 1) positive or 2) negative link. In case 1) the switch at stage $i+1$ reached by the positive link is $(j+2^i) \in S_{i+1}$ and, from Corollary 4.1, rerouting can be done through switch $(j-2^i) \in S_{i+1}$. In case 2) the switch at stage $i+1$ reached by the negative link is $(j-2^i) \in S_{i+1}$ and, from Corollary 4.1, rerouting can be done through switch $(j+2^i) \in S_{i+1}$. Let the switch at stage $i+1$ on the rerouting path be $w_{0/n-1}$. The state bits $b_{n-(i+1)/n-1}$ remain intact (equal to 0's) because it corresponds to having every switch from stage $i+1$ to $n-1$ remain in state C so that the IADM network from stage $i+1$ to $n-1$ can emulate the ICube network from stage $i+1$ to $n-1$. Thus, the bits l, $i+1 \le l \le n-1$, of the label of a switch on the rerouting path are $w_{l/n-1}$. From Lemma 2.1, bits 0 to $l-1$, $1 \le l \le i+1$, of the label of a switch on a path to destination $d_{0/n-1}$ must be $d_{0/l-1}$. Hence, the switch on the rerouting path from stage $i+1$ to $n-1$ has label $d_{0/l-1}w_{l/n-1}$, $i+1 \le l \le n-1$.

Next consider the case where the blockage of $j \in S_i$ is a straight link blockage or a double nonstraight link blockage so that backtracking is necessary. There are two subcases for each type of blockage: a) the nonstraight link found in backtracking is a negative link and b) it is a positive link. Here only subcase a) of the straight link blockage is considered; the other cases can be dealt with similarly. From the proof of Corollary 4.2 (case a) only), the switch on the rerouting path is $(j+2^l) \in S_l$, $i-k \le l \le i$. The switch of stage $i+1$ on the rerouting path is $j \in S_{i+1}$ if $b'_{n+i} = 0$ and $j \in S_{i+1}$ is an odd_i switch or if $b'_{n+i} = 1$ and $j \in S_{i+1}$ is an $even_i$ switch, and is $(j+2^{i+1}) \in S_{i+1}$ if $b'_{n+i} = 0$ and $j \in S_{i+1}$ is an $even_i$ switch or if $b'_{n+i} = 1$ and $j \in S_{i+1}$ is an odd_i switch. The identification of switches on the rerouting path from stage $i+1$ to $n-1$ is done as in the case of a nonstraight link blockage described above.

The blocked link can be represented by the two switches joined by the link. Since every switch on the original routing path and the rerouting paths can be easily identified as described above, it can be readily determined whether or not the blocked link is on the current path.

In summary, for both SDT schemes, the binary representation of the destination address can be used directly as the routing tag. In the SSDT scheme, rerouting tags are not needed and in the TSDT scheme, rerouting tags result from simple bit complementing operations. In terms of complexity of the computation for a rerouting tag, the SSDT scheme and the TSDT scheme for one instance of nonstraight link blockage require time × space complexity $O(1)$; an improvement over previous proposed schemes [9] dealing with rerouting for a nonstraight link blockage that require time × space complexity $O(\log N)$. In [10] a single-stage look-ahead scheme for rerouting of a straight link blockage was proposed; it requires use of two's complement to compute the positive and negative dominant tags so that the scheme has time × space complexity of $O(\log N)$. Note that the single-stage look-ahead rerouting scheme is valid only for some cases of the straight link blockage; it cannot be applied to any case of the straight link blockage. From Corollary 4.2, k-stage backtracking is needed for a straight link blockage and k bits of the state bits need to be changed; thus, the complexity of the TSDT scheme for a nonstraight link is $O(k)$. If only single-stage backtracking (corresponds to single-stage look-ahead) is necessary, rerouting can be done dynamically and the complexity is $O(1)$, an improvement over the scheme in [10].

V. A Universal Rerouting Algorithm for Multiple Blockages

The TSDT scheme can be applied to not only one instance of some blockage, but also can be applied repetitively each time a new blockage is encountered as the message propagates along. This section considers the derivation of an algorithm to deal with any case of multiple blockages. The backtracking schemes proposed in Corollary 4.2 find a rerouting path for a straight link blockage and a double nonstraight link blockage. Nevertheless, it is possible that blockages also exist on the rerouting path; then further backtracking to a lower-order stage is needed. Since this phenomenon can recur, repeated backtracking may be necessary due to blockages on the rerouting paths. The algorithm BACKTRACK described next performs iterated backtracking to find an alternate routing path. It underlies a universal rerouting algorithm (called REROUTE) to be shown later that can find a routing path, if there exists any, to bypass multiple blockages in the network.

The inputs to algorithm BACKTRACK are the current routing path P, the stage number i where a blockage occurs, and state bits $b'_{n/2n-1}$ representing path P. The algorithm returns updated values of the state bits $b'_{n/2n-1}$ which specify a rerouting path that is blockage-free from stage 0 to stage i if such a rerouting path exists, or returns FAIL if the blockages on the current routing path and the rerouting paths eliminate the possibility of communication between the source and the destination. It is assumed that the blockage on the original routing path at stage i is a straight link blockage or a double nonstraight link blockage and $j \in S_i$ is the switch whose output links are the blocked links. Informal explanations for

the algorithm will be given following the algorithm and the correctness proof of this algorithm can be found in [22].

Algorithm BACKTRACK (and REROUTE) presumes existence of the knowledge of all blockages in the network. The network controller is responsible for collecting this information and maintaining a global map of blockages, which is accessible to every sender of the messages in order to compute a path to avoid the blockages. In addition, since it may take several iterations before a blockage-free path can be found or it can be concluded that no blockage-free paths exist, the sender of the message needs to maintain and update the locations of switches on the rerouting path in each iteration.

Algorithm BACKTRACK $(P, i, b'_{n/2n-1})$

0: Let q be the stage number where a blockage occurs. $q \leftarrow i$.

1: P = the current routing path. Backtrack on path P from stage q to find a nonstraight link. If no nonstraight link exists at any preceding stage, return(FAIL); otherwise assign to r the stage number where the first nonstraight output link is found.

2: If the nonstraight link at stage r on the routing path is $+2^r$, assign flag *linkfound* value 0; if it is -2^r, assign *linkfound* value 1.

3: If *linkfound* $= 0$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} d_{r/q-1} \cdot b'_{n+q/2n-1}$;
if *linkfound* $= 1$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} \bar{d}_{r/q-1} \cdot b'_{n+q/2n-1}$.

4a: This step applies only when the blockage at stage q on path P is a straight link blockage.
If *linkfound* $= 0$, set $b'_{n+q} = d_q$; if $((j - 2^q) \in S_q, (j - 2^{q+1}) \in S_{q+1})$ is blocked, change b'_{n+q} to $\bar{d}_q$; furthermore, if $((j - 2^q) \in S_q, j \in S_{q+1})$ is also blocked, return(FAIL). If *linkfound* $= 1$, set $b'_{n+q} = \bar{d}_q$; if $((j+2^q) \in S_q, (j+2^{q+1}) \in S_{q+1})$ is blocked, change b'_{n+q} to d_q; furthermore, if $((j + 2^q) \in S_q, j \in S_{q+1})$ is also blocked, return(FAIL).

4b: This step applies only when the blockage at stage q on path P is a double nonstraight link blockage.
If $((j - 2^q) \in S_q, (j - 2^q) \in S_{q+1})$ is blocked for *linkfound* $= 0$, or $((j + 2^q) \in S_q, (j + 2^q) \in S_{q+1})$ is blocked for *linkfound* $= 1$, return(FAIL).

5: Let $\hat{Q}$ denote the part of the rerouting path (specified by the tag in step 3) from stage $r + 1$ to q from step 3. If *linkfound* $= 0$, $\hat{Q} = ((j - 2^{r+1}) \in S_{r+1}, \cdots, (j - 2^{q-1}) \in S_{q-1}, (j - 2^q) \in S_q)$; if *linkfound* $= 1$, $\hat{Q} = ((j + 2^{r+1}) \in S_{r+1}, \cdots, (j + 2^{q-1}) \in S_{q-1}, (j + 2^q) \in S_q)$.
If a blockage occurs on path $\hat{Q}$, return(FAIL).

6: If *linkfound* $= 0$, and $((j-2^r) \in S_r, (j-2^{r+1}) \in S_{r+1})$ is blocked, or if *linkfound* $= 1$ and $((j + 2^r) \in S_r, (j + 2^{r+1}) \in S_{r+1})$ is blocked, go to step 7; else return$(b'_{n/2n-1})$.

7: $j \leftarrow j + 2^r$, $q \leftarrow r$.

8: Backtrack on path P from stage q to find a nonstraight link. If no nonstraight link exists at any preceding stage, return(FAIL); otherwise assign to r the stage number where the first nonstraight output link is found.

9: If *linkfound* $= 0$ and the nonstraight link at stage r is -2^r, or if *linkfound* $= 1$ and the nonstraight link at stage r is $+2^r$, return(FAIL).

10: If *linkfound* $= 0$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} d_{r/q-1} \cdot b'_{n+q/2n-1}$;
if *linkfound* $= 1$, $b'_{n/2n-1} \leftarrow b'_{n/n+r-1} \bar{d}_{r/q-1} \cdot b'_{n+q/2n-1}$.
Go to step 4b.

Step 0 is the initialization step. From Theorems 3.3 and 3.4, an alternate path exists for avoiding a straight link blockage or a double nonstraight link blockage if and only if there exists a nonstraight link at some stage preceding stage r; step 1 of the algorithm searches backward for such a nonstraight link. If not found, it results in premature termination of the algorithm, reflecting the fact that no alternate paths for rerouting exist. Step 2 is used to differentiate the cases when the nonstraight link at stage r found in the first backtracking is a positive link and when it is a negative link; flag *linkfound* is assigned 0 for the former and 1 for the latter. If a nonstraight link exists at some stage preceding the blockages, in step 3, Corollary 4.2 is applied to find the stage bits specifying the rerouting path; cases a) and b) in Corollary 4.2 correspond to *linkfound* $= 1$ and *linkfound* $= 0$, respectively, and q and r correspond to i and $i - \kappa$, respectively.

Steps 4a and 4b deal with the link blockage at stage q on the rerouting path computed in step 3. If the blockage of a switch at stage q on path P is a straight link, the possible rerouting links at stage q are two nonstraight links. In step 4a the default link is a negative link if *linkfound* $= 0$ and a positive link if *linkfound* $= 1$. If the default link is blocked, step 4a attempts to reroute the message through the other nonstraight link. If both nonstraight links are blocked, there exist no blockage-free paths. Step 4b applies if the blockage of a switch at stage q on path P is a double nonstraight link blockage. The rerouting path must use a straight link at stage q. If it is also blocked, no blockage-free path exists.

Step 5 checks blockages from stage $r + 1$ to stage $q - 1$ on the rerouting path; if any blockage falls on $\hat{Q}$, there exists no blockage-free path. In step 6, if the blockage falls in the link of stage r on the rerouting path, further backtracking is necessary. Otherwise (no blockages on the rerouting path), the algorithm terminates with the state bits specifying the rerouting path. Step 7 updates the stage number q and the switch label j where a blockage on the rerouting path occurs, initiating a new iteration of backtracking. Step 8 is the same as step 1, searching backward at lower-order stages again for a nonstraight link. Step 9 of the algorithm dictates that if the encountered nonstraight link in the first iteration of backtracking is a positive (or negative) link, the nonstraight link found in each subsequent iteration of backtracking must be also a positive (or negative) link; otherwise no blockage-free paths exist. If the condition in step 9 is satisfied, step 10, which is the same as step 3, computes a rerouting path. After the rerouting path is found, the algorithm returns to step 4b, to check for further blockages on the rerouting path.

For each source/destination pair, a link on some routing path for the source/destination pair is called a *participating*

nk. As a direct result of Theorem 3.2, the set of participating utput links of a switch is composed of either its straight utput link or both of its nonstraight output links, but never ll of them. So the output link blockages of a switch, for given source/destination pair, can only be a straight link lockage, a nonstraight link blockage, or a double nonstraight nk blockage. Algorithm BACKTRACK deals with the first nd third kind of blockages, and the second kind of blockge can be overcome by applying Corollary 4.1. Algorithm BACKTRACK and Corollary 4.1 can be used to form a universal algorithm capable of rerouting messages when multiple lockages exist in the IADM network. This algorithm, called REROUTE, returns state bits $b'_{n/2n-1}$ specifying a blockage-ree rerouting path if one exists, or returns FAIL otherwise.

Algorithm REROUTE $(P, b'_{n/2n-1})$

0: P = the original routing path.
   $b_{n/2n-1}$ = the routing tag specifying the original routing path.
   $b'_{n/2n-1}$ = the rerouting tag specifying the rerouting path.
   $b'_{n/2n-1} \leftarrow b_{n/2n-1}$.
1: Let i be the smallest stage number such that there exists a blockage at stage i on path P. If no blockages occur on path P, return$(b'_{n/2n-1})$.
2: If the blockage at stage i on path P is a nonstraight link blockage and the other nonstraight link is not blocked, apply Corollary 4.1 to find state bits $b'_{n/2n-1}$ and go to step 4.
3: $b'_{n/2n-1} \leftarrow$ BACKTRACK$(P, i, b'_{n/2n-1})$.
4: Q = the rerouting path specified by state bits $b'_{n/2n-1}$. $P \leftarrow Q$ and go to step 1.

Step 0 is the initialization step. At the end of each iteration, blockage-free path from stage 0 to stage i is found. Then new iteration starts and i is given a new value in order to nd a path avoiding the blockages at a higher-order stage. The nly terminating conditions for algorithm REROUTE are that return of FAIL from step 3 indicating that no blockage-ee paths exist and the return from step 1 indicating a lockage-free path is found. Algorithm REROUTE is executed eratively to evade blockages from lower-order to higher-order ages. The correctness of this algorithm follows from the rrectness of algorithm BACKTRACK and Corollary 4.1.

VI. Permutation Routing and Cube Subgraphs of the IADM Network

The results discussed so far are a consequence of the istence of spare nonstraight links in addition to the ICube twork embedded in the IADM network. This section pursues is issue further by showing that there exist multiple distinct bgraphs in the IADM network, each called a *cube subgraph*, at are isomorphic to the ICube network. Two cube subgraphs e considered to be distinct if they differ in at least one link. s mentioned in the Introduction of this paper, the cube-type tworks have been studied extensively in the literature and own to be topologically equivalent. Together with results m these studies, the knowledge of how to identify cube bgraphs can help the understanding of the capabilities of the IADM network and be useful for permutation routing in the IADM network.

Since each switch can be in state C or $\overline{C}$, there are as many as $2^{N \cdot n} (= N^N)$ network states, although each does not necessarily generate a unique permutation. Setting a switch to a certain state indicates that one of its nonstraight output links can be used for routing (i.e., it is *active*) while the other cannot. Thus, each network state can be associated with a subgraph of the IADM network which contains only the active links. When all switches in the IADM network are set to state C, the IADM network functions as an ICube network; this network state corresponds to a cube subgraph. The constructive derivation of a lower bound for the number of cube subgraphs of the IADM network uses the two basic ideas discussed in the next paragraphs.

Since $+2^{n-1} \equiv -2^{n-1} \bmod N$, $C_{n-1}(j, t_{n-1}) = \overline{C}_{n-1}(j, t_{n-1})$, i.e., the state of each switch of stage $n-1$ is irrelevant in the sense that any switch at stage $n-1$ is always connected to the same two switches at stage n. Consequently, given any cube subgraph, there exist $(2^N - 1)$ subgraphs isomorphic to it which differ only in their choices of the nonstraight link $+2^{n-1}$ or -2^{n-1} at stage $n-1$. Therefore, the total number of distinct cube subgraphs is given by the product of 2^N and the number of distinct subgraphs of the IADM network from stage 0 to stage $n-2$ that are isomorphic to the same stages in the ICube network.

The calculation of the number of subgraphs in the first $n-1$ stages uses an idea similar to that proposed in [5] for reconfiguring the DR network so that it performs as a Generalized Cube network. All switches of the IADM network are logically relabeled by adding a constant x, $0 \le x \le N-1$ to the original labels, i.e., switch j becomes $j' = j + x$. By setting each switch to be an $even_i$ or odd_i switch according to its new label and having all switches be in state C, a cube subgraph results for each relabeling. However, of the N possible subgraphs, only $N/2$ are distinct as far as the first $n-1$ stages are concerned. This result is stated in Theorem 6.1 and the proof appears in [22]. A graphical interpretation of cube subgraph isomorphism for an IADM network of size $N = 8$ is illustrated in Fig. 8. In Fig. 8, each physical switch j acts as a logical switch $j' \equiv (j+1) \bmod 8$. The isomorphism to the ICube network can be easily visualized by moving switch 7 to the top of each stage as shown in the figure. Notice that setting some switch to state C according to its logical label may be equivalent to setting the switch to state $\overline{C}$ according to its original label. For instance, switch $0 \in S_0$ (logical label 1) is set to state $\overline{C}$ in Fig. 8.

Theorem 6.1: There exist at least $N/2 \cdot 2^N$ distinct cube subgraphs in the IADM network.

In order to reconfigure the IADM network to one of its cube subgraphs, each switch of stage i, for $0 \le i \le n-2$, needs to know the ith bit of its logical label. This can be done by sending the same logical label to every switch in the same row at system reconfiguration time. Each switch is set as being an odd_i or $even_i$ switch by examining the ith bit of the logical label. All switches operate in state C according to its logical label with the exception of those at stage $n-1$ for which different states correspond to different subgraphs.

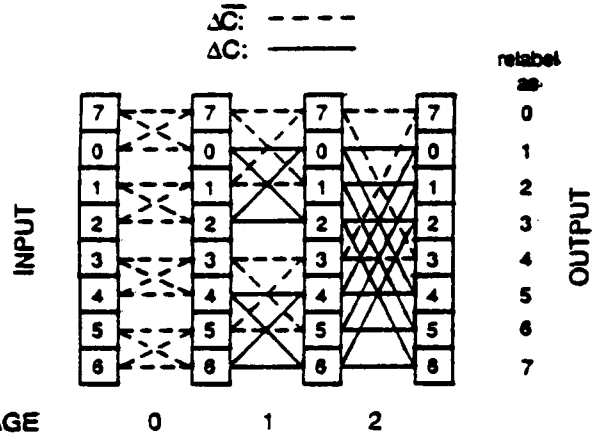


Fig. 8. A cube subgraph generated by relabeling each switch j to $(j+1) \bmod 8$ for an IADM network of size $N = 8$.

The results of this section can be used in different ways. One usage is in characterizing a class of permutations performable by the IADM network. Permutations passable by the ICube network are discussed in [15] and adaptable from [6]. Thus, the IADM network can perform all of these permutations plus the same set of permutations with a given x added to both the same source and destination labels, $0 \le x < N/2$. Another use of the results of this section is that the IADM network can pass the permutations performable by the ICube network when the ICube network embedded in the IADM network experiences nonstraight link failures. This is done by incorporating a reconfiguration function in the system that reassigns each switch j to $(j+x)$ and reconfiguring the IADM network to a corresponding cube subgraph which does not include the faulty nonstraight links. In [21] it is shown that any of the cube-type networks can pass the permutations performable by the others by incorporating appropriate reconfiguration functions. By the same token, the IADM network with a nonstraight link fault can also pass the permutations performable by the cube-type networks by including these reconfiguration functions in the system.

VII. Concluding Remarks

One of the main contributions of this paper is the identification of destination tag routing schemes for the IADM network. They are simpler and more efficient than previously known approaches, thus requiring less complex switches and reducing message communication delays due to routing overhead. In the SSDT scheme rerouting can be done when nonstraight links fail and in the TSDT scheme both the straight and double nonstraight link blockages can be avoided. As for the SSDT scheme, routing and rerouting are transparent to the source and only negligible hardware and time are used by each switch for routing and rerouting purpose. These are considerable advantages over previously proposed schemes which do not use destination tags and require extra hardware or delays of $O(\log N)$ complexity instead of $O(1)$. In addition, previous works all deal only with certain types of blockages. Based on the TSDT scheme, a universal rerouting algorithm is derived, which is capable of avoiding any combination of multiple blockages if there exists a blockage-free path and indicating absence of such a path if there exists none. The rerouting capabilities of the new schemes can be readily used for fault-tolerance and load balancing purposes since they adequately exploit the redundancy available in the IADM network.

Another contribution of this paper is the constructive derivation of a lower bound on the number of cube subgraphs of the IADM network. While it was previously known that the ICube network is a subgraph of the IADM network, this paper shows that there exist at least $N/2 \cdot 2^N$ distinct cube subgraphs. This, combined with previous multistage cube network studies, can help characterize some of the permutations performable by the IADM network. As other use of the subgraph analysis, it is shown how to reconfigure the IADM network under nonstraight link faults to pass the cube-admissible permutations.

Perhaps the most fundamental contribution of this paper is that of the network state model used for the IADM and the ICube networks. The essence of this model is in the recognition that the routing action of each switch is conceptually dependent on its position in the network (topological information), its state (functional information), and the destination of the message (routing information). Topological information is fixed and, when using destination tags, the same can be said of routing information for a given message destination. Consequently, the routing path is solely determined by the state of the network. These basic concepts are applicable to networks other than those considered in this paper; the state model can help devise new designs, solve routing problems, and understand relationships among networks.

References

[1] D. Agrawal, "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-32, pp. 637–648, July 1983.

[2] K. E. Batcher, "The Flip network in STARAN," in *Proc. 1976 Int. Conf. Parallel Processing*, Aug. 1976, pp. 65–71.

[3] T.-Y Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vols C-23, pp. 309–318, Mar. 1974.

[4] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," in *Proc. 1st Annu. Symp. Comput. Architecture*, Dec. 1973, pp. 21–28.

[5] M. Jeng and H. J. Siegel, "Design and analysis of dynamic redundancy networks," *IEEE Trans. Comput.*, vol. 37, pp. 1019–1029, Sept. 1988.

[6] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145–1155, Dec. 1975.

[7] D. Lee and K. Y. Lee, "Control algorithms for the augmented data manipulator network," in *Proc. 1986 Int. Conf. Parallel Processing*, Aug. 1986, pp. 123–130.

[8] M. Malek and W. W. Myre, "A description method of interconnection networks," *IEEE Tech. Committee Distrib. Process.*, Quart. vol. 1, pp. 1–6, Feb. 1981.

[9] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202–1214, Dec. 1982.

[10] ——, "Performance and fault tolerance improvements in the inverse augmented data manipulator network," in *Proc. 9th Annu. Symp. Comput. Architecture*, Apr. 1982, pp. 63–72.

[11] ——, "Evaluation of cube and data manipulator networks," *J. Parallel Distributed Comput.*, vol. 2, no. 1, pp. 79–107, Feb. 1985.

[12] K. Padmanabhan and D. H. Lawrie, "A class of redundant path multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-32, pp. 1099–1108, Dec. 1983.

[13] D. S. Parker and C. S. Raghavendra, "The Gamma network: A multiprocessor interconnnection network with redundant paths," in *Proc. 9th Annu. Symp. Comput. Architecture*, Apr. 1982, pp. 73–80.

[14] ——, "The Gamma network," *IEEE Trans. Comput.*, vol. C-33, pp. 367–373, Apr. 1984.
[15] M. C. Pease, III, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458–473, May 1977.
[16] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, second ed. New York: McGraw-Hill, 1990.
[17] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," in *Proc. 5th Annu. Symp. Comput. Architecture*, Apr. 1978, pp. 223–229.
[18] H. J. Siegel and R. J. McMillen, "The multistage cube: A versatile interconnection network," *IEEE Comput. Mag.*, vol. 14, pp. 65–76, Dec. 1981.
[19] A. Varma and C. S. Raghavendra, "On permutations passible by the Gamma network," *J. Parallel Distributed Comput.*, vol. 3, no. 1, pp. 72–91, Mar. 1986.
[20] C-L. Wu and T-Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694–702, Aug. 1980.
[21] ——, "The reverse-exchange interconnection network," *IEEE Trans. Comput.*, vol. C-29, pp. 801–811, Sept. 1980.
[22] D. Rau, J. A. B. Fortes, and H. J. Siegel, "Destination tag routing techniques based on a state model for the IADM network," Tech. Rep. TR-EE 87-39, School of Electrical Engineering, Purdue Univ., West Lafayette, IN, 47907, Oct. 1987.

Darwen Rau received the Ph.D. degree in electrical engineering from Purdue University, West Lafayette, IN, in 1988, the M.S. degree in mechanical engineering from the University of Massachusetts, and the B.S. degree in industrial engineering from National Tsing Hua University, Taiwan.

He is a member of the Technical Staff at AT&T Bell Laboratories, Naperville, IL. Prior to joining Bell Laboratories, he was a Senior Software Engineer at Digital Equipment Corporation. His research interests include interconnection networks, ISDN, software maintenance, and multimedia.

Jose A. B. Fortes (S'80–M'84) for a photograph and biography, see the February 1992 issue of this TRANSACTIONS, p. 206.

Howard Jay Siegel (M'77–SM'82–F'90) received two B.S. degrees from MIT, and the M.A., M.S.E., and Ph.D. degrees from Princeton University.

He is a Professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue University. He has coauthored over 150 technical papers, coedited four volumes, and written one book. He does consulting, has prepared two "continuing education" eleven-hour video tape courses, and has given tutorials for various organizations in the U.S.A., Europe, and Israel, all in the area of parallel computing.

Dr. Siegel has served as program Co-Chair of the 1983 International Conference on Parallel Processing, and as General Chair of the 3rd International Conference on Distributed Computing Systems and the 15th Annual International Symposium on Computer Architecture. He is Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and Program Chair of Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computations. His current research focuses on interconnection networks and the use and design of the PASM reconfigurable parallel computer system (a prototype of which is supporting active experimentation). He is a member of the Eta Kappa Nu and Sigma Xi honorary societies.

REFERENCE NO. 6

Shang, W. and Fortes, J. A. B., "On Mapping Uniform Dependence Algorithms into Lower Dimensional Processor Arrays," *IEEE Transactions on Parallel and Distributed Systems*, Volume 3, Number 3, May 1992, pp. 350-363.

Note - Two-dimensional processors with faulty elements can be easily reconfigured as one-dimensional arrays. However, it is necessary to remap the original algorithm into an array of smaller dimension than that of the original array. This paper presents a mapping methodology which addresses this problem.

On Time Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays

Weijia Shang, *Member, IEEE*, and Jose A. B. Fortes, *Member, IEEE*

Abstract—**Most existing methods of mapping algorithms into processor arrays are restricted to the case where n-dimensional algorithms, or algorithms with n nested loops, are mapped into $(n-1)$-dimensional arrays. However, in practice, it is interesting to map n-dimensional algorithms into $(k-1)$-dimensional arrays where $k < n$. For example, many algorithms at bit level are at least four-dimensional (matrix multiplication, convolution, LU decomposition, etc.) and most existing bit level processor arrays are two-dimensional. A *computational conflict* occurs if two or more computations of an algorithm are mapped into the same processor and the same execution time. In this paper, based on the Hermite normal form of the mapping matrix, necessary and sufficient conditions are derived to identify mappings without computational conflicts. These conditions are used to find time mappings of n-dimensional algorithms into $(k-1)$-dimensional arrays, $k < n$, without computational conflicts. For some applications, the mapping is time-optimal.**

Index Terms—**Bit level algorithm, conflict-free, nested loops, optimal time mapping, processor array.**

I. Introduction

MOST existing methods of mapping algorithms into processor arrays are restricted to the cases where n-dimensional algorithms, or algorithms with n nested loops, are mapped into $(n-1)$-dimensional processor arrays [2]–[13]. For example, the three-dimensional matrix multiplication algorithm is usually mapped into a two-dimensional processor array by these methods [10], [21], [32]. This paper considers time mappings of n-dimensional algorithms into $(k-1)$-dimensional, $k < n$, processor arrays. Procedures are proposed to find time mappings (space mappings are assumed to be given) without *computational conflicts*, which means no two or more computations of the algorithm are mapped into the same processor and execution time.

Algorithms under consideration in this paper are nested loops with regular data dependence structures. Such algorithms can be modeled by a *uniform dependence algorithm* (J, D). Set J is the *index set*, or *iteration space*. Each element in J is an n-tuple integral column vector (called *iteration vector* or *index vector*). Matrix D is the *dependence matrix* where each column is a dependence vector. If computation in one iteration depends on the computation in another iteration, this dependence is represented by the vector difference of the iteration vectors corresponding to these two iterations. All dependences are assumed to be *uniform* which means the dependence relation exists between two iterations as long as the vector difference between these two iterations is equal to the vector representing that dependence relation. This algorithm model can be easily related to similar models and concepts in [1]–[13], [19] and several other works. Uniform dependence algorithms occur frequently in digital signal and image processing and scientific computing applications [36].

Examples of two-dimensional bit level processor arrays include GAPP [33], DAP [34], MPP [35], MP-1 [31] etc. Many bit level algorithms are four or five dimensional, such as matrix multiplication, convolution, LU decomposition, etc. How to automatically map these algorithms into two-dimensional bit level arrays is still a problem [28]. That is why in practice it is interesting to develop a method to map n-dimensional algorithms into $(k-1)$-dimensional processor arrays with $k < n$. The work reported in this paper was motivated by the implementation of RAB (Reconfiguration Algorithm for Bit level code) [26], an experimental tool which maps a class of algorithms programmed in "C" into bit level arrays. In the RAB approach, algorithms are first expanded into bit level, and second, the dependence relations are analyzed and the algorithm is uniformized. Then the global optimal solution, which maps often a four or five dimensional bit level algorithm into a two-dimensional bit level processor array, is to be found.

Several attempts have been made to try to map algorithms into lower dimensional systolic arrays [15], [22], [23], [25]. In particular, important steps toward a formal solution to this problem were made in [23]. Based on the Lamport hyperplane transformation model [13], a procedure was proposed to find mappings of three-dimensional algorithms into one-dimensional, or linear systolic arrays, without computational conflicts and data link collisions. Five conditions were given to guarantee the correctness of the mapping. The first condition ensures that dependence relations among different computations of the algorithm are respected; the second condition is about computational conflicts; the third and fourth conditions deal with the number of shift registers on links and the data

Manuscript received May 1, 1990; revised June 11, 1991. This work was supported in part by Louisiana Education Quality Support Fund under Contract LEQSF(1991-93)-RD-A-42, by the National Science Foundation under Grant DCI-8419745, and by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization, and was administered through the Office of Naval Research under Contracts 00014-85-k-0588, 00014-88-k-0723, and 00014-90-J-1483.

W. Shang is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9107409.

travel directions; and the fifth condition is to avoid data link collisions. The concept of data link collisions and the conditions to avoid such collisions are introduced in this work. Detection of computational conflicts is basically by a heuristic analysis of all computations of the algorithm and the optimality of the mapping is not guaranteed. A suboptimal solution for the matrix multiplication algorithm was found in [23] by which the total execution time is $(3+\mu)\mu+1$ for multiplying two $(\mu+1)\times(\mu+1)$ matrices. In [22], further results are reported in mapping n-dimensional algorithms into $(k-1)$-dimensional processor arrays. A suboptimal solution for the reindexed transitive closure algorithm [17], [23] was found by the proposed procedure in [22] by which the total execution time is $\mu(2\mu+3)+1$ where μ is the problem size.

This paper describes a method of finding time mappings of n-dimensional uniform dependence algorithms into $(k-1)$-dimensional arrays, $k<n$, without any computational conflicts. Based on the Hermite normal form of the mapping matrix, necessary and sufficient conditions are derived to guarantee a conflict-free mapping. These conditions are used to formulate the problem of finding time-optimal and conflict-free mappings as an integer programming problem. The mapping is optimal for some applications. Compared to the method in [22] and [23], the main contributions of this paper are the closed form necessary and sufficient conditions for conflict-free mappings based on the Hermite normal form of the mapping matrix. In addition, by using these conditions, the problem of identifying time-optimal and conflict-free mappings is formulated as an integer programming optimization problem. For some algorithms such as the matrix multiplication algorithm and the transitive closure algorithm, the integer programming formulation can be further converted to linear programming problems. In Section V, the method proposed in this paper is used to find optimal solutions for the matrix multiplication algorithm and the reindexed transitive closure algorithm which improve the total execution time of $(3+\mu)\mu+1$ in [23] and $\mu(2\mu+3)+1$ in [22] to $(2+\mu)\mu+1$ and $\mu(\mu+3)+1$, respectively, where μ is the problem size.

This paper is organized as follows. Section II presents basic terminology and definitions, introduces the concept of computational conflicts, and provides statements of problems addressed in this paper. Section III discusses a simple case to illustrate different aspects of, and provide insight into, the conflict-free mapping problem. Section IV discusses the conflict-free mapping problem in general. Section V presents an optimization procedure and integer programming problem formulations which can be used to find mappings without any computational conflicts. Section VI concludes this paper and points out some future work.

II. Terminology and Definitions

Throughout this paper, *sets*, *matrices*, and *row vectors* are denoted by capital letters, *column vectors* are represented by lower case symbols with an overbar and *scalars* correspond to lower case letters. The *transpose* of a vector $\bar{v}$ is denoted $\bar{v}^T$. The vector $\bar{0}$ denotes the row or column vector whose entries are all zeroes. The dimensions of vector $\bar{0}$ and whether it denotes a row or column vector are implied by the context in which they are used. The symbol I denotes the identity matrix. The rank and the determinant of matrix A are denoted $\text{rank}(A)$ and $\det A$, respectively. The set of integers, the set of nonnegative integers, and the set of positive integers are denoted Z, N, and N^+, respectively. The empty set is denoted $\emptyset$. The notations $|C|$ and $|\alpha|$ represent the cardinality, or the number of elements, of set C and the absolute value of scalar α, respectively. Let $\bar{v}$ and $\bar{u}$ be two vectors. Then $\bar{v}\geq\bar{u}$ means every component of $\bar{v}$ is greater than or equal to the corresponding component of $\bar{u}$. Finally, if x is an element of a set S, the notation $x\in S$ is used and this notation is also used to indicate that a column vector $\bar{m}_j$ (or row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j\in M$ ($M_i\in M$) means $\bar{m}_j$ (M_i) is a column (row) vector of matrix M.

Algorithms of interest in this paper are the so-called uniform dependence algorithms defined as follows.

Definition 2.1 (Uniform dependence algorithm): A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\bar{j})=g_{\bar{j}}(v(\bar{j}-\bar{d}_1),v(\bar{j}-\bar{d}_2),\cdots,v(\bar{j}-\bar{d}_m)) \qquad (2.1)$$

where

1) $\bar{j}\in J\subset Z^n$ is an index point (a column vector), J is the *index set*, or *iteration space*, of the algorithm and $n\in N^+$ is the *algorithm dimension*, or the number of components of $\bar{j}$;
2) $g_{\bar{j}}$ is the computation indexed by $\bar{j}$, i.e., a single-valued function computed "at point $\bar{j}$" in a *single unit of time*;
3) $v(\bar{j})$ is the value computed "at $\bar{j}$," i.e., the result of computing the right-hand side of (2.1) and
4) $\bar{d}_i\in Z^n, i=1,\cdots,m, m\in N$ are *dependence vectors*, also called *dependencies*, which are constant (i.e., independent of $\bar{j}\in J$); the matrix $D=[\bar{d}_1,\cdots,\bar{d}_m]$ is called the *dependence matrix*.

The class of uniform dependence algorithms is a simple extension of the class of algorithms described by uniform recurrence equations [1]. The main difference is that uniform dependence algorithms allow for different functions to be computed (in a unit of time) at different points of the index set. From a practical viewpoint, uniform dependence algorithms can be easily related to programs where 1) a single statement appears in the body of a multiply nested loop and 2) the indexes of the variable in the left-hand side of the statement differ by a constant from the corresponding indexes in each reference to the same variable in the right-hand side. Alternative computations can occur in each iteration as a result of a single conditional statement as long as data dependencies do not change. Nested loop programs with multiple statements can also use the techniques of this paper together with the alignment method discussed in [14] and [24]. Uniform dependence algorithms occur frequently in scientific computing and digital signal processing applications.

For the purpose of finding time-optimal and conflict-free mappings, only structural information of the algorithm, i.e., the index set J and the dependence matrix D, is needed. Therefore, a uniform dependence algorithm with index set

J and dependence matrix D is herein characterized simply by the pair (J, D). Each index vector $\vec{j} \in J$ corresponds to a computation; and computation $\vec{j}$ depends on computations $\vec{j} - \vec{l}_i \in J$, $i = 1, \cdots, m$. It is assumed that, as in Definition 2.1, the letters n and m always denote the *algorithm dimension* and the number of dependence vectors, respectively.

Many models have been proposed to map algorithms into processor arrays. The *linear algorithm transformation* method proposed in [2], [12], and [32] is used in this paper and stated as follows.

Definition 2.2 (Linear algorithm transformation): A *linear algorithm transformation* maps an n-dimensional uniform dependence algorithm into a $(k-1)$-dimensional processor array according to the mapping:

$$\tau : J \to Z^k, \tau(\vec{j}) = T\vec{j}, \forall \vec{j} \in J \tag{2.2}$$

where $T = \begin{bmatrix} S \\ \Pi \end{bmatrix} \in Z^{k \times n}$ is the *mapping matrix*, $S \in Z^{(k-1) \times n}$ is the *space mapping matrix*, and $\Pi \in Z^{1 \times n}$ is the *time mapping vector*, or *linear schedule vector*. The computation indexed by $\vec{j} \in J$ is executed at time $\Pi\vec{j}$ and at processor $S\vec{j}$. The mapping τ must satisfy the following conditions:

1) $\Pi D > \vec{0}$.
2) $SD = PK$ where $P \in Z^{(k-1) \times r}$ is the matrix of interconnection primitives of the target machine, $K \in Z^{r \times m}$ is such that

$$\sum_{j=1}^{r} k_{ji} \le \Pi \vec{d}_i, \qquad i = 1, \cdots, m. \tag{2.3}$$

3) $\forall \vec{j}_1, \vec{j}_2 \in J$, if $\vec{j}_1 \neq \vec{j}_2$, then $\tau(\vec{j}_1) \neq \tau(\vec{j}_2)$, or $T\vec{j}_1 \neq T\vec{j}_2$.
4) The rank of T is equal to k, or $\text{rank}(T) = k$.

Condition 1 in Definition 2.2 preserves the partial ordering induced by the dependence vectors. If this condition is satisfied, then computation indexed by $\vec{j} \in J$ is scheduled to execute only after the executions of computations indexed by $\vec{j} - \vec{d}_i \in J$, $i = 1, \cdots, m$ because $\Pi D > \vec{0}$, and therefore the dependence relation is respected.

The matrix of interconnection primitives P describes the connection links of processors in the array. For an array with each processor connected to its four nearest east, south, west, and north neighbors, it has four interconnection primitives $[0, 1]^T$, $[0, -1]^T$, $[1, 0]^T$, and $[-1, 0]^T$, and matrix $P = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{bmatrix}$. Condition 2 in Definition 2.2 guarantees that the space mapping can be implemented in a fixed systolic architecture with interconnection primitive matrix P. The summation in the left-hand side of the inequality in (2.3) is the number of times of the usage of interconnection primitives to pass the datum caused by the dependence vector $\vec{d}_i$ from the source to the destination. The item in the right is the time units between the source usage and the destination usage of that datum. Assuming it takes one time unit for a datum to travel one interconnection primitive, the inequality must be satisfied to have the datum arrive before it is used. Condition 2 in Definition 2.2 may not be required when a new processor array is designed specially for the algorithm. It is required only when the algorithm is to be mapped into a processor array with a fixed interconnection structure.

Condition 3 is for avoiding computational conflicts because if $\tau(\vec{j}_1) = \tau(\vec{j}_2)$, then the computations indexed by $\vec{j}_1$ and $\vec{j}_2$ are mapped into the same processor and time and a conflict occurs. Condition 4 guarantees that the algorithm is to be mapped into a $(k-1)$-dimensional array but not a q-dimensional array, $q < k-1$. When $\text{rank}(T) = q+1 < k$, there are exactly $q+1$ linearly independent rows in T, and all other rows of T are linear combinations of these $q+1$ linearly independent rows. Let T' be the matrix consisting of these $q+1$ linearly independent rows; then T can be transformed linearly to T' which means the algorithm is actually mapped into a q-dimensional processor array.

More constraints on the mapping τ are possible for some implementation requirements. In addition, different constraint forms from those in Definition 2.2 for the same implementation requirement can be used. For example, in [23], the inequality in (2.3) is required to be an equality which means data must arrive right at the time of their usage and are not allowed to arrive before the usage. Also, in [23], constraints to avoid data link collisions are considered.

Because the execution of any computation needs one time unit as defined in Definition 2.1, the total execution time by the linear schedule vector Π is as follows:

$$t = \max\{\Pi(\vec{j}_1 - \vec{j}_2) : \vec{j}_1, \vec{j}_2 \in J\} + 1. \tag{2.4}$$

For a class of practical algorithms, the loop bounds are constants. This kind of algorithm is characterized by the *constant-bounded index set* defined as

$$J = \{[j_1, \cdots, j_n]^T : 0 \le j_i \le \mu_i, j_i \in Z, \mu_i \in N^+, \\ i = 1, \cdots, n\} \tag{2.5}$$

where zero and μ_i correspond to the lower and upper bounds of the ith loop, respectively. Upper bounds μ_i, $i = 1, \cdots, n$, are called *problem size variables*. To simplify the problem, this paper is restricted to the algorithms with constant-bounded index sets. This assumption is summarized as follows.

Assumption 2.1: In this paper, the index sets under consideration are assumed to be constant-bounded defined formally by (2.5).

Some other kinds of algorithms can be transformed into algorithms with constant-bounded index sets by a linear mapping of the index sets [12]. For an algorithm with a constant-bounded index set, because

$$\max\{\Pi(\vec{j}_1 - \vec{j}_2) : \vec{j}_1, \vec{j}_2 \in J\} \\ = [|\pi_1|, \cdots, |\pi_n|]([\mu_1, \cdots, \mu_n]^T - \vec{0}), \tag{2.6}$$

the total execution time t in (2.4) can be simplified to

$$t = 1 + \sum_{i=1}^{n} |\pi_i| \mu_i. \tag{2.7}$$

From (2.7), the vector Π which minimizes the objective function t in (2.7) is such that the absolute values of its entries $|\pi_i|$, $i = 1, \cdots, n$ are as small as possible and with some constraints satisfied. In other words, if the absolute value of

any one of the entries of the optimal Π is reduced by one, then the resulting vector is not a valid linear schedule vector. This conclusion is also indicated in [10] and [11] and is summarized n the following theorem.

Theorem 2.1 [10], [11]: The total execution time described in (2.4) is a monotonically increasing function of $|\pi_i|$, $i = 1, \cdots, n$, the absolute values of entries of vector Π.

A conflict occurs if two or more computations are mapped into the same processor and the same execution time. That is, for two distinct index points $\bar{j}_1$ $\bar{j}_2 \in J$, if $T\bar{j}_1 = T\bar{j}_2$, then there is a conflict. For the case where $k = n$, or T is a square matrix, that $\text{rank}(T) = n$ guarantees a conflict-free mapping because $T\bar{j}_1 = T\bar{j}_2$ if and only if $\bar{j}_1 = \bar{j}_2$. For the case where $k < n$, even when $\text{rank}(T) = k$, or matrix T has full row rank, there is at least one nonzero vector $\bar{\gamma}$ such that $T\bar{\gamma} = 0$. Let $\bar{j}_1 = \bar{j}_2 + \bar{\gamma}$, then $T\bar{j}_1 = T\bar{j}_2$. If both $\bar{j}_1$ and $\bar{j}_2$ belong to the index set, then the computations indexed by $\bar{j}_1$ and $\bar{j}_2$, respectively, are mapped to the same processor and the same execution time and a conflict occurs. Therefore, it is much more difficult to find a mapping without conflicts when $k < n$ than for the case when $k = n$.

One possible way to avoid conflicts is to find the mapping matrix T such that, for any arbitrary index point $\bar{j} \in J$ and any $\bar{\gamma}$ that is a nonzero integral solution of equation $T\bar{\gamma} = 0$, $\bar{j} + \bar{\gamma}$ does not belong to the index set J. This concept is illustrated by Fig. 1 which shows a two-dimensional index set $J = \{[j_1, j_2]^T: 0 \le j_1, j_2 \le 4, j_1, j_2 \in Z\}$. If $\bar{\gamma}$ is $\bar{\gamma}_1 = [1, 1]^T$, then index points $\bar{j} = \bar{0}$ and $\bar{j} + \bar{\gamma}_1 = [1, 1]^T$ both belong to index set J and computations indexed by $[0, 0]^T$, $[1, 1]^T$, $[2, 2]^T, \cdots, [4, 4]^T$ will be mapped into the same processor and the same execution time. Therefore, there is at least one conflict. However, if $\bar{\gamma}$ is $\bar{\gamma}_2 = [3, 5]^T$, there will be no conflict at all because for any arbitrary $\bar{j} \in J$, $\bar{j} + \bar{\gamma}_2 \notin J$. Intuitively, if vector $[3, 5]^T$ is drawn with one end at $[0, 0]^T$ (or at any other index point of the index set), then the other end is out of the index set and vector $[3, 5]^T$ does not meet any integer points in the index set. Therefore, the mapping with this $\bar{\gamma}$ is conflict-free. To describe these concepts formally, the following definitions are introduced.

Definition 2.3 (Conflict vector, feasible and nonfeasible conflict vectors, and conflict-free mapping matrix): Given an algorithm (J, D) and a mapping matrix $T \in Z^{k \times n}$, an integral column vector $\bar{\gamma} = [\gamma_1, \ldots, \gamma_n]^T$ is a *conflict vector* of the mapping matrix T if and only if $T\bar{\gamma} = 0$ and $\gcd^1(\gamma_1, \cdots, \gamma_n) = 1$. If for any arbitrary index point $\bar{j} \in J$, $\bar{j} + \bar{\gamma} \notin J$, then $\bar{\gamma}$ is a *feasible conflict vector*. If there exists at least one index point $\bar{j} \in J$ such that $\bar{j} + \bar{\gamma} \in J$ then $\bar{\gamma}$ is called a *nonfeasible conflict vector*. If all the conflict vectors are feasible, then this mapping matrix T is *conflict-free*.

Example 2.1: Consider a four-dimensional algorithm (J, D) where

$$J = \{\bar{j} : \bar{j} \in Z^4, 0 \le j_i \le 6, i = 1, \cdots, 4\}.$$

Assume that this algorithm is to be mapped into a one-

[1] $\gcd(a_1, \cdots, a_n)$ denotes the greatest common divisor of integers $a_1, \cdots, a_n$.

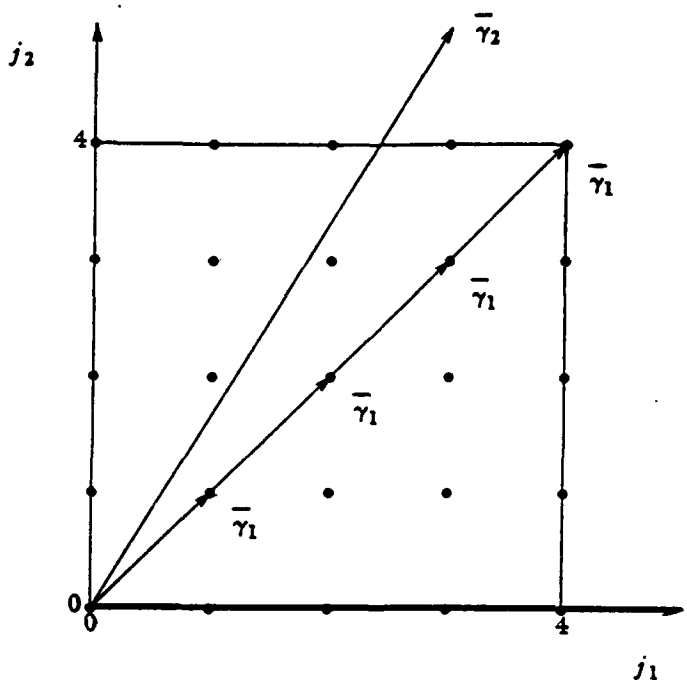


Fig. 1. Nonfeasible conflict vector $\bar{\gamma}_1$ and feasible conflict vector $\bar{\gamma}_2$. Vector $\bar{\gamma}_2$ does not meet any integral points inside the index set.

dimensional, or linear, processor array and one possible mapping matrix is

$$T = \begin{bmatrix} 1 & 7 & 1 & 1 \\ 1 & 7 & 1 & 0 \end{bmatrix}. \tag{2.8}$$

Consider the following solutions of $T\bar{\gamma} = \bar{0}$: $\bar{\gamma}_1 = [0, 1, -7, 0]^T$, $\bar{\gamma}_2 = [7, -1, 0, 0]^T$ and $\bar{\gamma}_3 = [1, 0, -1, 0]^T$. Clearly, $T\bar{\gamma}_1 = T\bar{\gamma}_2 = T\bar{\gamma}_3 = \bar{0}$ and their greatest common divisors of their entries are unity. So $\bar{\gamma}_1$, $\bar{\gamma}_2$, and $\bar{\gamma}_3$ are conflict vectors of mapping matrix T. However, vector $[2, 0, -2, 0]^T$ is also a solution of equation $T\bar{\gamma} = \bar{0}$ but is not a conflict vector of mapping matrix T because the greatest common divisor of its entries is not unity. Conflict vectors $\bar{\gamma}_1$ and $\bar{\gamma}_2$ are feasible because it can be checked that for any arbitrary index point $\bar{j} \in J$, $\bar{j} + \bar{\gamma}_i \notin J$, $i = 1, 2$. Conflict vector $\bar{\gamma}_3$ is not feasible because for the index point $\bar{j} = [0, 0, 1, 0]^T \in J$, $\bar{j} + \bar{\gamma}_3 = [1, 0, 0, 0]^T \in J$. Therefore, T is not conflict-free. □

For algorithms with constant-bounded index sets, the following theorem describes the common characteristics of feasible conflict-vectors.

Theorem 2.2: For algorithms with constant-bounded index sets defined by (2.5), a mapping matrix T is conflict-free if and only if for each of its conflict vectors $\bar{\gamma} = [\gamma_1, \cdots, \gamma_i, \cdots, \gamma_n]^T$ there exists an entry γ_i such that $|\gamma_i| > \mu_i$.

Proof: ($\Rightarrow$). Because T is conflict-free, all the conflict vectors of T are feasible. Now suppose that $\bar{\gamma}$ is a conflict vector of T and $|\gamma_i| \le \mu_i$, $i = 1, \cdots, n$. Consider the index point $\bar{j} = [j_1, \cdots, j_n]^T$ where $j_i = 0$ if $\gamma_i \ge 0$ and $j_i = -\gamma_i$ if $\gamma_i < 0$. Both $\bar{j}$ and $\bar{j} + \bar{\gamma}$ belong to the index set J defined by (2.5) because $|\gamma_i| \le \mu_i$, $i = 1, \cdots, n$. By Definition 2.3, $\bar{\gamma}$ is not feasible which is contrary to the assumption. Therefore,

for each of the conflict vectors $\bar{\gamma}$, there must exist an entry γ_i such that $|\gamma_i| > \mu_i$.

($\Leftarrow$). Let $\bar{\gamma}$ be a conflict vector of mapping matrix T and consider an arbitrary index point $\bar{j}$ belonging to the index set defined by (2.5). Let $\bar{j}' = \bar{j} + \bar{\gamma} = [j_1', \cdots, j_n']^T$. Because there exists an entry γ_i of $\bar{\gamma}$ such that $|\gamma_i| > \mu_i$ and $\mu_i \geq j_i \geq 0$, $j_i' = j_i + \gamma_i > \mu_i$ if $\gamma_i > 0$ and $j_i' = j_i + \gamma_i < 0$ if $\gamma_i < 0$. In both cases, $\bar{j}'$ is not in the index set J and $\bar{\gamma}$ is feasible. This implies that T is conflict-free. □

In practice, it is interesting to find optimal conflict-free mappings with respect to different criteria and based on different assumptions. To achieve this, one has to identify first all feasible mappings that are conflict-free. Then it is possible to choose an optimal one with respect to a certain criterion from these conflict-free mappings. The criterion could be the total execution time for the algorithm, the VLSI area taken to implement this algorithm including the number of processors and the length of the wiring, or the combination of the total execution time and the VLSI area. Two problems are addressed in this paper and are formulated below. The first is about identifying conflict-free mappings and the second is about finding optimal time mappings where space mappings are given.

Problem 2.1 (Conflict-free mapping problem): Given an n-dimensional uniform dependence algorithm and a $(k-1)$-dimensional processor array, find necessary and sufficient conditions for mapping matrix $T \in Z^{k \times n}$ to be conflict-free.

Problem 2.2 (Time-optimal and conflict-free mapping problem): Given an n-dimensional uniform dependence algorithm (J, D) and a space mapping matrix $S \in Z^{(k-1)\times n}$, find an integral row vector $\Pi^o \in Z^{1\times n}$ which minimizes

$$f = \max\{\Pi(\bar{j}_1 - \bar{j}_2) : \bar{j}_1, \bar{j}_2 \in J\}$$

$$\text{subject to } \begin{cases} \Pi D > \bar{0} \\ \sum_{j=1}^{r} k_{ji} \leq \Pi \bar{d}_i, \quad \text{where } SD = PK \\ \text{rank}(T) = k \\ T = \begin{bmatrix} S \\ \Pi \end{bmatrix} \text{ is conflict-free.} \end{cases}$$

In Problem 2.2, the objective function f differs by one from the total execution time t in (2.4). Clearly, f is minimized if and only if t is minimized. P and K are as defined in Definition 2.2. The solution of a special case of Problem 2.1 is discussed in Section III, and the general case is discussed in Section IV followed by the discussion of Problem 2.2.

III. Necessary and Sufficient Conditions for Conflict-Free Mapping Matrix $T \in Z^{(n-1)\times n}$

This section discusses the solution of Problem 2.1, or how to identify all conflict-free mapping matrices $T \in Z^{(n-1)\times n}$ that map n-dimensional algorithms into $(n-2)$-dimensional processor arrays. This simplest case can illustrate and give an intuitive understanding of different aspects of the conflict-free mapping problem so the reader can follow the general discussion in the next section more easily. Practical applications are the mapping of four-dimensional convolution algorithm at bit level [26] into a two-dimensional systolic array and the mapping of the three-dimensional matrix multiplication algorithm into a linear systolic array [23].

Let $\Pi \in Z^{1\times n}$, $S \in Z^{(n-2)\times n}$, and $\text{rank}(S) = n-2$. Consider the following equation

$$T\bar{\gamma} = \bar{0} \text{ or } \begin{bmatrix} S \\ \Pi \end{bmatrix} \bar{\gamma} = \bar{0}. \tag{3.1}$$

Let us first assume that $\text{rank}(T) = n-1$. Later in this section, conditions on Π are given to guarantee that $\text{rank}(T) = n-1$. Clearly, there is only one linearly independent solution of (3.1). Without loss of generality, let $T = [B, \bar{b}]$ where B contains the first $n-1$ columns of matrix T, $\text{rank}(B) = n-1$, and $\bar{b}$ is the last column of T. Also, let B^* and $\det B$ be the adjugate matrix and determinant of matrix B, respectively [18, p. 170]. Then all solutions of (3.1) can be expressed as

$$\bar{\gamma} = \lambda \begin{bmatrix} -B^*\bar{b} \\ \det B \end{bmatrix} = \lambda \begin{bmatrix} f_1(\pi_1, \cdots, \pi_n) \\ f_2(\pi_1, \cdots, \pi_n) \\ \cdots \\ f_n(\pi_1, \cdots, \pi_n) \end{bmatrix}. \tag{3.2}$$

where λ is a constant.

If the first nonzero entry of a conflict vector is assumed to be positive (this implies no loss of generality), then for the mapping matrix $T \in Z^{(n-1)\times n}$, there is only one unique conflict vector (otherwise, $-\bar{\gamma}$ would also be a conflict vector). This unique conflict vector $\bar{\gamma}$ is expressed by (3.2) where λ is such that $\bar{\gamma}$ is integral, its entries are relatively prime, and the first nonzero entry is positive. According to Theorem 2.2 if this unique conflict vector is feasible, then the corresponding mapping is conflict-free. In addition, if Π is such that there exists a nonzero entry $f_i(\pi_1, \cdots, \pi_n), 1 \leq i \leq n$, then $\text{rank}(T) = n-1$ because $f_i(\pi_1, \cdots, \pi_n)$ is the determinant of the submatrix of T consisting of all columns except the ith one of T. These facts are summarized in the following theorem.

Theorem 3.1 (Necessary and sufficient condition 1): Let $\bar{\gamma}$ be defined in (3.2) where the constant λ in (3.2) is such that $\bar{\gamma}$ is integral, its entries are relatively prime, and the first nonzero entry is positive. Then mapping matrix $T \in Z^{(n-1)\times n}$ is feasible if and only if vector $\bar{\gamma}$ is feasible. The rank of matrix T is $n-1$ if and only if there exists a nonzero entry $f_i(\pi_1, \cdots, \pi_n), 1 \leq i \leq n$.

Proof: First, it is shown as follows that there is only one conflict vector if the first nonzero entry of the conflict vector is assumed to be positive. Suppose there are two conflict vectors $\bar{\gamma}_1$ and $\bar{\gamma}_2$ whose first nonzero entries are positive. Because there is only one linearly independent solution of (3.2), $\bar{\gamma}_1$ and $\bar{\gamma}_2$ are linearly dependent. Thus, $\bar{\gamma}_2 = c\bar{\gamma}_1$, where c is a constant. If $c = 1$, then $\bar{\gamma}_1 = \bar{\gamma}_2$; if $c = -1$, then the first entry of one of the vectors is not positive; if c is a nonintegral rational number, then $\bar{\gamma}_2$ is nonintegral because the greatest common divisor of entries of $\bar{\gamma}_1$ is one; and if $c > 1$ is integral, the greatest common divisor of $\bar{\gamma}_2$ is greater than unity. Therefore, in all the cases discussed above, $\bar{\gamma}_2$ is not a distinct conflict vector whose first entry is positive. So there is only one such conflict vector of mapping matrix T and, therefore, T is feasible if and only if $\bar{\gamma}$ is feasible.

It is trivial to show that if $\text{rank}(T) = n-1$, then there is a nonzero entry $f_i(\pi_1, \cdots, \pi_n)$ because otherwise, $\text{rank}(T) = n$. Now suppose there exists a nonzero entry $f_i(\pi_1, \cdots, \pi_n)$.

Let

$$B^* = \begin{bmatrix} B_{11} & B_{21} & \cdots & B_{(n-1),1} \\ B_{12} & B_{22} & \cdots & B_{(n-1),2} \\ \cdots & \cdots & \cdots & \cdots \\ B_{1,(n-1)} & B_{2,(n-1)} & \cdots & B_{(n-1),(n-1)} \end{bmatrix} \quad (3.3)$$

where B_{ij}, $i, j = 1, \cdots, n-1$, are the cofactors of matrix B [18, p. 165]. Clearly, $f_i = [B_{1i}, \cdots, B_{(n-1),i}]\bar{b} = B_{1i}s_{1,n} + \cdots + B_{(n-2),i}s_{(n-2),n} + B_{(n-1),i}\pi_n$, $1 \le i \le (n-1)$. With little thought, it can be seen that f_i is the determinant of matrix B with the ith column being replaced by $\bar{b}$ [18, p. 165] which is a submatrix of T. Therefore, there is a submatrix of T whose determinant is nonzero which means $\text{rank}(T) = n-1$. □

According to Theorem 2.2, the unique conflict vector $\bar{\gamma}$ in (3.2) is feasible if and only if the absolute value of one of its entries is greater than a certain value. Therefore, given a mapping matrix T, to see if it is conflict-free or not, (3.2) can be used.[2] Later in Section V, (3.2) is used to formulate problem 2.2 as an integer programming problem. If functions f_i in (3.2), $i = 1, \cdots, n$, are linear, then the formulation is possibly an integer linear programming problem. In the following, it is shown that if the space mapping matrix S is given, functions f_i in (3.2), $i = 1, \cdots, n$, are linear functions of π_j, $j = 1, \cdots, n$.

Proposition 3.2: Functions f_i, $i = 1, \cdots, n$ in (3.2) are linear functions of π_j, $j = 1, \cdots, n$.

Proof: Let B^* be defined as in (3.3). Clearly, $f_i = [B_{1i}, \cdots, B_{(n-1),i}]\bar{b} = B_{1i}s_{1,n} + \cdots + B_{(n-2),i}s_{(n-2),n} + B_{(n-1),i}\pi_n$, $1 \le i \le (n-1)$. Cofactors B_{li}, $1 \le l \le n-2$, are the linear functions of π_j, $j = 1, \cdots, n-1$ because B_{li} is the determinant of the submatrix of B obtained from removing the lth row and the ith column of matrix B. Thus, $B_{li}s_{l,n}$ are linear functions of π_j, $j = 1, \cdots, n-1$. Cofactor $B_{(n-1),i}$ is independent of π_j, $j = 1, \cdots, n$ because it is the determinant of the submatrix of B obtained by removing the ith column and the $(n-1)$th row which is $\bar{b}$. Thus, $B_{(n-1),i}b_{n-1} = B_{(n-1),i}\pi_n$ is a linear function of π_n. Therefore, f_i, $i = 1, \cdots, n-1$ are linear functions of π_j, $j = 1, \cdots, n$. The last entry $f_n = \det B$ is also a linear function of π_j, $j = 1, \cdots, n-1$ because it is the determinant of matrix B. □

Example 3.1: Consider algorithm (J, D) used in [23] where dependence matrix D and index set J are as follows.

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$J = \{\bar{j} : \bar{j} \in Z^3, 0 \le j_i \le \mu, i = 1, \cdots, 3\}. \quad (3.4)$$

Actually, this uniform dependence algorithm can model the matrix multiplication algorithm. How the dependence matrix and the index set are derived from the Fortran code of the matrix multiplication algorithm is shown in [21], [23], and [2]. Let the matrix multiplication algorithm compute $C = AB$, where $C = [c_{ij}]$, $A = [a_{ij}]$, and $B = [b_{ij}]$. By [23], dependence vectors $\bar{d}_1$, $\bar{d}_2$, and $\bar{d}_3$ are induced by B, A, and C, respectively. In other words, the computation indexed by $\bar{j}$ needs data A from index point $\bar{j} - \bar{d}_2$, B from index point $\bar{j} - \bar{d}_1$, and C from index point $\bar{j} - \bar{d}_3$. If the space mapping matrix is chosen as the one used in [23] $S = [1, 1, -1]$, then mapping matrix T and its conflict vector $\bar{\gamma}$ are as follows.

$$T = \begin{bmatrix} 1 & 1 & -1 \\ \pi_1 & \pi_2 & \pi_3 \end{bmatrix}, \quad \bar{\gamma} = \lambda \begin{bmatrix} -\pi_2 - \pi_3 \\ \pi_1 + \pi_3 \\ \pi_1 - \pi_2 \end{bmatrix}. \quad (3.5)$$

It is clear that $T\bar{\gamma} = \bar{0}$. If Π is chosen such that $-\pi_2 - \pi_3 \neq 0$ or $\pi_1 + \pi_3 \neq 0$ or $\pi_1 - \pi_2 \neq 0$, then $\text{rank}(T) = n - 1 = 2$. □

Example 3.2: Consider another algorithm (J, D) used in [22] where dependence matrix D and index set J are as follows:

$$D = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \end{bmatrix}$$

$$J = \{\bar{j} : \bar{j} \in Z^3, 0 \le j_i \le \mu, i = 1, \cdots, 3\}. \quad (3.6)$$

This uniform dependence algorithm can model the reindexed transitive closure algorithm. How the dependence matrix and the index set are derived from the Fortran code of the transitive closure algorithm is shown in [17] and [23]. If the space mapping matrix is chosen as the one used in [22] $S = [0, 0, 1]$, then mapping matrix T and its conflict vector $\bar{\gamma}$ are as follows.

$$T = \begin{bmatrix} 0 & 0 & 1 \\ \pi_1 & \pi_2 & \pi_3 \end{bmatrix}, \quad \bar{\gamma} = \lambda \begin{bmatrix} \pi_2 \\ -\pi_1 \\ 0 \end{bmatrix}. \quad (3.7)$$

It is clear that $T\bar{\gamma} = \bar{0}$. If Π is chosen such that $\pi_2 \neq 0$ or $\pi_1 \neq 0$ then $\text{rank}(T) = n - 1 = 2$. □

IV. General Case—Necessary and Sufficient Conditions for Conflict-Free Mappings

This section discusses and presents the solution to the general case of Problem 2.1, i.e., it provides necessary and sufficient conditions for conflict-free mappings where n-dimensional algorithms are mapped into $(k-1)$-dimensional processor arrays. In these mappings, $T \in Z^{k \times n}$, $T = \begin{bmatrix} S \\ \Pi \end{bmatrix}$, $\Pi \in Z^{1 \times n}$ and $S \in Z^{(k-1) \times n}$.

Consider the equation

$$T\bar{\gamma} = \bar{0} \text{ or } \begin{bmatrix} S \\ \Pi \end{bmatrix}\bar{\gamma} = \bar{0}. \quad (4.1)$$

If $\text{rank}(T) = k$, then there are $n - k$ linearly independent solutions of (4.1). Let $\bar{\gamma}_1, \cdots, \bar{\gamma}_{n-k}$ be the linearly independent integral solutions of (4.1), whose entries are relatively prime, then all solutions $\bar{\gamma}$ of (4.1) can be represented as linear combinations of the $n-k$ linear independent vectors as follows

$$\bar{\gamma} = \lambda_1\bar{\gamma}_1 + \cdots + \lambda_{n-k}\bar{\gamma}_{n-k}. \quad (4.2)$$

Clearly, $\bar{\gamma}_1, \cdots, \bar{\gamma}_{n-k}$ are conflict vectors of T.

In general, the mapping matrix T has more than $n - k$ conflict vectors when $k < n - 1$ because a linear combination of these $n - k$ conflict vectors may be a different integral vector whose entries are relatively prime and therefore another conflict vector of T. This new conflict vector may or may not be feasible. Thus, unlike the mapping matrix $T \in Z^{(n-1) \times n}$

[2] The unique conflict vector (a vector in the nullspace of T) can be obtained by other methods such as Gaussian elimination.

described in Section III, it is not guaranteed that all conflict vectors of T are feasible even if the $n-k$ linearly independent solutions $\bar{\gamma}_i$, $i = 1, \cdots, n-k$, of equation $T\bar{\gamma} = \bar{0}$ are all feasible. This is illustrated by the following example.

Example 4.1: Consider the four-dimensional algorithm of Example 2.1 and the mapping matrix T in (2.8). Let $\bar{\gamma}_1 = [0, 1, -7, 0]^T$ and $\bar{\gamma}_2 = [7, -1, 0, 0]^T$. Clearly, $T\bar{\gamma}_1 = T\bar{\gamma}_2 = \bar{0}$, $\bar{\gamma}_1$, and $\bar{\gamma}_2$ are linearly independent, and they are feasible conflict vectors of T. Let $\bar{\gamma} = 1/7\bar{\gamma}_1 + 1/7\bar{\gamma}_2 = [1, 0, -1, 0]^T$. Vector $\bar{\gamma}$ is also a solution of equation $T\bar{\gamma} = \bar{0}$ and its entries are relatively prime. By Definition 2.3 and Theorem 2.2, $\bar{\gamma}$ is a nonfeasible conflict vector of T. Therefore, as mentioned above, for a given mapping matrix $T \in Z^{k\times n}$ with $k < n-1$, there are possibly more than $n-k$ conflict vectors, and T may not be conflict-free even if there are $n-k$ linearly independent feasible conflict vectors of T. □

From Example 4.1, an interesting observation is that one difficulty in making all conflict vectors of mapping matrix T feasible is that nonfeasible conflict vectors can result from rational linear combinations of the $n-k$ linearly independent feasible conflict vectors $\bar{\gamma}_1, \cdots, \bar{\gamma}_{n-k}$ like $\bar{\gamma} = 1/7\bar{\gamma}_1 + 1/7\bar{\gamma}_2$ in Example 4.1. Let us consider another way to select the $n-k$ linearly independent conflict vectors of T such that constants λ_i, $i = 1, \cdots, n-k$ in (4.2) must be integral in order for $\bar{\gamma}$ to be integral. To achieve this, the notion of the Hermite normal form is introduced.

Theorem 4.1 (Hermite normal form [29, p. 45]): Let $T \in Z^{k\times n}$ and $\text{rank}(T) = k$. Then there exists a unimodular[3] matrix $U \in Z^{n\times n}$ such that $TU = H = [L, 0]$ (0 denotes a zero-entry matrix) where $L \in Z^{k\times k}$ is a nonsingular and lower triangular matrix. Matrix H is called the *Hermite normal form* of T.

The definition of the Hermite normal form used here is slightly different from the one used conventionally and in [29], where each diagonal element of matrix L is required to be positive and be the maximum of all absolute values of elements in that same row. This is because for the purpose of this paper it is enough to know that matrix T can be transformed into a lower triangular matrix $[L, 0]$, by right multiplication of a unimodular matrix U.

For a given mapping matrix T, let H be the corresponding Hermite normal form and $T = HV$ where $V = U^{-1}$, $U = [\bar{u}_1, \cdots, \bar{u}_n]$ and $V = [\bar{v}_1, \cdots, \bar{v}_n]$. Then (4.1) can be rewritten as $HV\bar{\gamma} = \bar{0}$. Let $\bar{\beta} = V\bar{\gamma} = [\beta_1, \cdots, \beta_n]^T$ and $\bar{\gamma} = U\bar{\beta}$. Then the following statements are true.

Theorem 4.2:

1) $H\bar{\beta} = \bar{0}$ if and only if $\beta_1, \cdots, \beta_k$ are zero.
2) Vector $\bar{\gamma}$ is integral if and only if $\bar{\beta}$ is integral.
3) Vector $\bar{\gamma}$ is a conflict vector of mapping matrix T if and only if

$$\bar{\gamma} = [\bar{u}_{k+1}, \cdots, \bar{u}_n] \begin{bmatrix} \beta_{k+1} \\ \cdots \\ \beta_n \end{bmatrix} \tag{4.3}$$

where β_i, $i = k+1, \cdots, n$, are arbitrary integers which are not all zero and are relatively prime.

[3] A matrix is *unimodular* if and only if it is integral and the absolute value of its determinant is one.

Proof: 1) Because $H = [L, 0]$ and $H\bar{\beta} = L[\beta_1, \cdots, \beta_k]^T$ where $L \in Z^{k\times k}$ is a nonsingular lower triangular matrix, $\beta_1, \cdots, \beta_k$ are zero if and only if $H\bar{\beta} = \bar{0}$.

2) By definition, a matrix is unimodular if and only if it is integral and the absolute value of its determinant is unity. So U is unimodular means that matrix $V = U^{-1}$ is also unimodular. Therefore, $\bar{\gamma}$ is integral implies that $\bar{\beta}$ is integral and vice versa.

3) By Theorem 4.2 1) and 2), all integral solutions $\bar{\gamma}$ of equation $T\bar{\gamma} = \bar{0}$ are represented by (4.3) where β_i, $i = k+1, \cdots, n$, have to be arbitrary integers because nonintegral values of β_i, $i = k+1, \cdots, n$ result in a nonintegral vector $\bar{\gamma}$. Next it is shown that the greatest common divisor of β_i, $i = 1, \cdots, n$ is unity if and only if the greatest common divisor of γ_i, $i = 1, \cdots, n$ is unity. Suppose $\gcd(\beta_1, \cdots, \beta_n) = 1$ and $\gcd(\gamma_1, \cdots, \gamma_n) = c > 1$. Then $\bar{\gamma} = c\bar{\gamma}'$ where $\bar{\gamma}'$ is integral and its entries are relatively prime. Because $\bar{\beta} = V\bar{\gamma} = cV\bar{\gamma}'$ where, obviously $V\bar{\gamma}' \in Z^n$, the greatest common divisor of β_i, $i = 1, \cdots, n$ is at least $c > 1$. This is contrary to the assumption. So, the greatest common divisor of β_i, $i = 1, \cdots, n$ is unity implies the greatest common divisor of γ_i, $i = 1, \cdots, n$ is unity. With similar reasoning, the reverse can be shown. Therefore, the β_i, $i = k+1, \cdots, n$, in (4.3) have to be relatively prime integers, otherwise the greatest common divisor of entries of vector $\bar{\gamma}$ is greater than one, which is not a conflict vector by Definition 2.3. □

What Theorem 4.2 implies is that all conflict vectors of mapping matrix T can be represented by (4.3) where $\beta_{k+1}, \cdots, \beta_n$ are arbitrary integers which are not all zero and are relatively prime. Notice that a nonintegral value of any one of the $\beta_{k+1}, \cdots, \beta_n$ results in a nonintegral vector $\bar{\gamma}$ according to Theorem 4.2. So in this representation, the case where a new conflict vector of T can be obtained by a nonintegral linear combination of the $n-k$ linearly independent solutions of (4.1) is avoided.

Example 4.2: The Hermite normal form of the mapping matrix T in (2.8) is

$$TU = H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

where

$$U = \begin{bmatrix} 1 & -1 & -1 & -7 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \text{ and } V = U^{-1} = \begin{bmatrix} 1 & 7 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

All conflict vectors of T are the integral combinations of the third and fourth columns of matrix U as follows:

$$\bar{\gamma} = \begin{bmatrix} -1 & -7 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \beta_3 \\ \beta_4 \end{bmatrix}$$

where β_3 and β_4 are integers which are not all zero and are relatively prime. □

So far, a better representation of all conflict vectors of T has been found which requires integral combinations of $n-k$ linearly independent conflict vectors of mapping matrix T. However, it is still not guaranteed that all conflict vectors are feasible unless matrix U satisfies some conditions. The

following five theorems describe these necessary and sufficient conditions for mapping matrix T to be conflict-free.

Theorem 4.3 (Necessary condition 2): Let v_{ij} be the entry of matrix V at the ith row and the jth column. If mapping matrix T is conflict-free, then at least one of the first k entries of each and every column of V must be nonzero, that is, the following condition holds.

$$\begin{aligned}&(v_{11} \neq 0 \vee v_{21} \neq 0 \vee \cdots \vee v_{k1} \neq 0) \wedge \\ &(v_{12} \neq 0 \vee v_{22} \neq 0 \vee \cdots \vee v_{k2} \neq 0) \wedge \\ &\cdots \\ &(v_{1n} \neq 0 \vee v_{2n} \neq 0 \vee \cdots \vee v_{kn} \neq 0). \end{aligned} \tag{4.4}$$

Proof: Let $\bar{\gamma}$ be an arbitrary conflict vector of mapping matrix T. If T is conflict-free, then $\bar{\gamma}$ is feasible. Feasible conflict vector $\bar{\gamma}$ has at least two nonzero entries $\gamma_i \neq 0$ and $\gamma_j \neq 0$, $i \neq j$. Otherwise, $\bar{\gamma}$ has only one nonzero entry which is unity because the greatest common divider of entries of $\bar{\gamma}$ is required to be one. Such a conflict vector is not feasible. Next, it is shown that $\bar{\gamma}$ has at least two nonzero entries if and only if (4.4) holds.

($\Leftarrow$). Suppose that $\bar{\gamma}$ has only one nonzero entry, then $\bar{\gamma} = [0, \cdots, 0, 1, 0, \cdots, 0]^T$. By assumption, $\bar{\beta} = V\bar{\gamma} = \bar{v}_i$, the ith column of matrix V. According to (4.4), there exists a nonzero element $v_{li} \in \{v_{1i}, \cdots, v_{ki}\}$ which means $\beta_l \neq 0$, $1 \leq l \leq k$. This is contrary to Theorem 4.2 1) that $\beta_i = 0$, $i = 1, \cdots, k$. Therefore, $\bar{\gamma}$ has at least two nonzero entries.

($\Rightarrow$). Suppose there exists a column $\bar{v}_i$ of matrix V whose first k entries are all zero, then $\bar{v}_i = [0, \cdots, 0, v_{k+1,i}, \cdots, v_{ni}]^T$. Let $\bar{\beta} = \bar{v}_i$. Because the first k entries of $\bar{v}_i$ are zero, $H\bar{\beta} = \bar{0}$ and $\bar{\gamma} = V^{-1}\bar{\beta}$ is a conflict vector of mapping matrix T. However, $\bar{\gamma} = V^{-1}\bar{v}_i = [0, \cdots, 0, 1, 0, \cdots, 0]^T$ whose entries are all zero except that the ith entry is unity. So, mapping matrix T has a conflict vector with only one nonzero entry which is contrary to the assumption. This means (4.4) holds. □

Theorem 4.4 (Necessary condition 3): If mapping matrix T is feasible, then $\bar{u}_{k+1}, \cdots, \bar{u}_n$ are feasible conflict vectors.

Proof: By definitions, $T\bar{u}_i = \bar{0}$, $i = k+1, \cdots, n$. Thus, $\bar{u}_{k+1}, \cdots, \bar{u}_n$ are conflict vectors of T. T is feasible implies $\bar{u}_i$, $i = k+1, \cdots, n$, must be feasible. □

Theorem 4.5 (Sufficient condition 4): Mapping matrix T is conflict-free if the following conditions are met.

There exist $i_1, \cdots, i_{n-k} \in \{1, \cdots, n\}$ such that

1)

$$\begin{aligned}&\gcd(u_{i_1,k+1}, u_{i_1,k+2}, \cdots, u_{i_1,n}) > \mu_{i_1} \\ &\gcd(u_{i_2,k+1}, u_{i_2,k+2}, \cdots, u_{i_2,n}) > \mu_{i_2} \\ &\cdots \\ &\gcd(u_{i_{n-k},k+1}, u_{i_{n-k},k+2}, \cdots, u_{i_{n-k},n}) > \mu_{i_{n-k}}\end{aligned}$$

2)

$$\det \begin{bmatrix} u_{i_1,k+1} & u_{i_1,k+2} & \cdots & u_{i_1,n} \\ u_{i_2,k+1} & u_{i_2,k+2} & \cdots & u_{i_2,n} \\ \cdots & \cdots & \cdots & \cdots \\ u_{i_{n-k},k+1} & u_{i_{n-k},k+2} & \cdots & u_{i_{n-k},n} \end{bmatrix} \neq 0.$$

Proof: Let $\bar{\gamma}$ be an arbitrary conflict vector of mapping matrix T represented by (4.3) where $\beta_{k+1}, \cdots, \beta_n$ are arbitrary integers which are not all zero and are relatively prime. Because

$$\det \begin{bmatrix} u_{i_1,k+1} & u_{i_1,k+2} & \cdots & u_{i_1,n} \\ u_{i_2,k+1} & u_{i_2,k+2} & \cdots & u_{i_2,n} \\ \cdots & \cdots & \cdots & \cdots \\ u_{i_{n-k},k+1} & u_{i_{n-k},k+2} & \cdots & u_{i_{n-k},n} \end{bmatrix} \neq 0$$

and $\beta_{k+1}, \cdots, \beta_n$ are not all zero, there exists $i \in \{i_1, \cdots, i_{n-k}\}$ such that

$$[u_{i,k+1}, u_{i,k+2}, \cdots, u_{i,n}] \begin{bmatrix} \beta_{k+1} \\ \beta_{k+2} \\ \cdots \\ \beta_n \end{bmatrix} = \gamma_i \neq 0. \tag{4.5}$$

According to condition 1), $\gcd(u_{i,k+1}, \cdots, u_{i,n}) = \alpha_i \geq \mu_i + 1$. If $|\gamma_i| < \mu_i + 1$, then α_i does not divide γ_i which means, according to [27], (4.5) has no integral solution and $\beta_{k+1}, \cdots, \beta_n$ are not all integral. Thus, $\bar{\gamma}$ is not integral and not a conflict vector. Therefore, it must be $|\gamma_i| \geq \mu_i + 1$. By Definition 2.3 and Theorem 2.2, $\bar{\gamma}$ is feasible and T is conflict-free. □

Theorem 4.6 (Sufficient condition 5 for $T \in Z^{(n-2)\times n}$): Mapping matrix $T \in Z^{(n-2)\times n}$ is conflict-free if the following conditions are met.

1) There exists $i \in \{1, \cdots, n\}$ such that $\gcd(u_{i,n-1}, u_{i,n}) \geq \mu_i + 1$.
2) Let β_{n-1} and β_n be relatively prime integers, not both zero and such that $\beta_{n-1}u_{i,n-1} + \beta_n u_{i,n} = 0$, there exists $j \in \{1, \cdots, n\}$, $j \neq i$, such that $|\beta_{n-1}u_{j,n-1} + \beta_n u_{j,n}| > \mu_j$.

Proof: Consider all integral values for β_{n-1} and β_n which are not both zero, are relatively prime, and $\beta_{n-1}u_{i,n-1} + \beta_n u_{i,n} \neq 0$. Let the corresponding conflict vectors be $\bar{\gamma}$. Because $\gamma_i \neq 0$ and $\gcd(u_{i,n-1}, u_{i,n}) = \alpha_i \geq \mu_i + 1$, $|\gamma_i| \geq \mu_i + 1$. Otherwise, α_i does not divide γ_i and equation $\beta_{n-1}u_{i,n-1} + \beta_n u_{i,n} = \gamma_i$ has no integral solution [27]. Therefore, for these values of β_{n-1} and β_n, the corresponding conflict vectors are feasible. Now let us consider the integral values for β_{n-1} and β_n which are not both zero, are relatively prime and $\beta_{n-1}u_{i,n-1} + \beta_n u_{i,n} = 0$. For the corresponding conflict vector $\bar{\gamma}$, because there exists $j \in \{1, \cdots, n\}$, $j \neq i$, such that $|\beta_{n-1}u_{j,n-1} + \beta_n u_{j,n}| > \mu_j$, $|\gamma_j|$ is greater than μ_j and $\bar{\gamma}$ is feasible. Therefore, mapping matrix T is conflict-free because all of its conflict vectors are feasible. □

Theorem 4.7 (Sufficient condition 6 for $T \in Z^{(n-2)\times n}$): Mapping matrix $T \in Z^{(n-2)\times n}$ is conflict-free if the following conditions are met.

1) There exists $i \in \{1, \cdots, n\}$ such that $u_{i,n-1} \cdot u_{i,n} \geq 0$ and $|u_{i,n-1} + u_{i,n}| > \mu_i$.
2) There exists $j \in \{1, \cdots, n\}$ such that $u_{j,n-1} \cdot u_{j,n} \leq 0$ and $|u_{j,n-1} - u_{j,n}| > \mu_j$.
3) $\bar{u}_n - 1$ and $\bar{u}_n$ are feasible conflict vectors.

Proof: Because $T \in Z^{(n-2)\times n}$, its conflict vectors are described by (4.3) where $k = n-2$. β_{n-1} and β_n are arbitrary integers that are not both zero and are relatively prime. Suppose conditions 1), 2), and 3) are met. Let us

consider the case where one of β_{n-1} and β_n is zero. Then the corresponding conflict vector $\bar{\gamma}$ equals either $\bar{u}_{n-1}$ or $\bar{u}_n$ and is feasible in either case because of condition 3). Let us consider the second case where $\beta_{n-1} \neq 0$ and $\beta_n \neq 0$ have the same sign and the corresponding conflict vector $\bar{\gamma}$. By condition 1), there exists $i \in \{1, \cdots, n\}$ such that $u_{i,n-1}, u_{i,n}$ have the same sign and $|u_{i,n-1}+u_{i,n}| > \mu_i$. Because u_{n-1} has the same sign as u_n and β_{n-1} has the same sign as β_n, $|\bar{\gamma}_i| = |\beta_{n-1}u_{i,n-1}+\beta_n u_{i,n}| = |\beta_{n-1}u_{i,n-1}| + |\beta_n u_{i,n}| \geq |u_{i,n-1} + u_{i,n}| > \mu_i$. Therefore, $\bar{\gamma}$ is feasible. Now let us consider the third case where $\beta_{n-1} \neq 0$ and $\beta_n \neq 0$ have opposite signs and the corresponding conflict vector $\bar{\gamma}$. According to condition 2), there exists $j \in \{1, \cdots, n\}$ such that $u_{j,n-1}, u_{j,n}$ have opposite signs and $|u_{j,n-1} - u_{j,n}| > \mu_j$. Because β_{n-1} and β_n have different signs, $|\gamma_j| = |\beta_{n-1}u_{j,n-1} + \beta_n u_{j,n}| \geq |u_{j,n-1} - u_{j,n}| > \mu_j$. Therefore, $\bar{\gamma}$ is feasible. □

Finding a closed form necessary and sufficient condition on conflict-free mappings for general case is still open. The following discussion on the mapping matrices $T \in Z^{(n-2)\times n}$ might provide some insight why this problem is difficult. Let U_i' denote the ith, $1 \leq i \leq n$, row of the $n \times 2$ matrix which has the last two columns of U as its columns. Let $C = \{[\beta_{n-1}, \beta_n]^T : |U_i'[\beta_{n-1}, \beta_n]^T| \leq \mu_i,\ i = 1, \cdots, n\}$. If C contains integral elements whose two entries are relatively prime and not all zero, then the corresponding mapping has computational conflicts. C is a convex polyhedron. Thus, to find closed form necessary and sufficient conditions on conflict-free mappings can be reduced to the closed form conditions to test if a convex polyhedron contains integral elements which is a difficulty open problem in integer programming problem area. One possible approach is to construct a minimal hypercube which contains the convex polyhedron C. Because the upper and lower bounds of the hypercube in each dimension are constant, it is easier to test if this hypercube contains integral elements which are in C. Based on this idea, instead of the closed form necessary and sufficient conditions, a procedure is developed to test if a mapping matrix is conflict-free or not. This procedure will be reported in a separate paper [37].

Multiplier matrix U is involved in most of the necessary and sufficient conditions for conflict-free mappings and the key point is to have matrix U satisfy certain conditions in order to make the mapping matrix T conflict-free. Given a mapping matrix T in numbers, there are several numerical methods to compute matrix U in polynomial time [20]. When the entries of T are symbols (variables), computing matrix U becomes more difficult. However, when the space mapping matrix S is known, it is possible to express matrix U as a function of Π. In [30], matrix U is expressed as a function of Π for the mapping matrix $T \in Z^{3\times 5}$.

V. Conflict-Free Mappings

Solutions to Problem 2.1 have been discussed in the above sections. In this section, Problem 2.2 and its solutions are discussed: that is, how to find the optimal vector Π° which contributes a conflict-free mapping matrix T and satisfies some other constraints. A procedure is presented which searches the solution space intelligently by employing the basic strategy in [10] and [11]. The integer linear programming formulation plus some heuristic is used to find optimal time mappings for the matrix multiplication algorithm and the reindexed transitive closure algorithm.

By Theorem 2.1, the total execution time increases monotonically with the sum of the weighted absolute values of entries of vector Π. Therefore, to find the solution of Problem 2.2, one simple way is to modify the method in [10] and [11] where candidates are enumerated in an increasing order of their total execution times. In the modified method, besides other considerations, each candidate should be checked to see if it contributes a conflict-free mapping. This method is described in the following procedure.

Procedure 5.1 (Finding optimal solution Π° of Problem 2.2):

Input: Algorithm (J, D) and the space mapping $S \in Z^{(k-1)\times n}$.

Output: An integral row vector Π° which is the solution of Problem 2.2.

Step 1: Set $l = 1$. $C_0 = \emptyset$.

Step 2: Construct a candidate set $C_l = \{\Pi : \sum_{i=1}^n |\pi_i|\mu_i \leq x_l \in N^+\}$. $C = C_l - C_l \cap C_{l-1}$. Let $|C| = c$.

Step 3: Sort and assign indices to all candidates in C such that $t(\Pi_i) \leq t(\Pi_j), 0 < i < j \leq c$, where $t(\Pi)$ is the function described in (2.7).

Step 4: Set $i = 1$.

Step 5: Consider candidate Π_i; set $T^i = \begin{bmatrix} S \\ \Pi_i \end{bmatrix}$ and check if Π_i meets the following conditions

a) $\Pi D > \bar{0}$.

b) $\text{rank}(T^i) = k$.

c) T^i is conflict-free.

d) $SD = PK$ and $\sum_{j=1}^r k_{ji} \leq \Pi \bar{d}_i, i = 1, \cdots, m$, where P and K are as defined in Definition 2.2.

Step 6: If Π_i meets all the three conditions above, then $\Pi^\circ = \Pi_i$ and stop. If Π_i does not meet any one of the conditions above, ignore this candidate, set $i = i + 1$.

Step 7: If $i > c$, set $l = l + 1, x_l = x_l + \alpha, \alpha \in N^+$ and go to Step 2. If $i \leq c$, go to Step 5.

To test if T^i is conflict-free or not in step 5, for mapping matrices $T \in Z^{(n-1)\times n}$, the necessary and sufficient conditions in Theorem 3.1 can be used. For other mapping matrices, either these necessary conditions in Section IV or the testing procedure in [37] can be used. The computational complexity of Procedure 5.1 is estimated at least $\Theta((2\mu+1)^n)$ where $\mu = \min\{\mu_i : i = 1, \cdots, n\}$. This is justified as follows. Consider one conflict vector $\bar{\gamma}$ of T and let S_i, $i = 1, \cdots, k-1$ be the sum of the absolute values of entries of the ith row of space mapping matrix S. As indicated in [37], $\sum_{i=1}^n |\pi_i| \geq ((\max\{|\gamma_1|, \cdots, |\gamma_n|\})^{n-k})/(\prod_{i=1}^{k-1} S_i)$. For $\bar{\gamma}$ to be feasible, at least, $\max\{|\gamma_1|, \cdots, |\gamma_n|\} \geq \mu + 1$. Let $\prod_{i=1}^{k-1} S_i = c$ be constant. Then $\sum_{i=1}^n |\pi_i| \geq \mu^{n-k}/c$ must be satisfied. Thus, some of the possible values Procedure 5.1 at least should try for π_i, $i = 1, \cdots, n$ are $0, \pm 1, \cdots, \pm\mu$. Because for each entry π_i, $1 \leq i \leq n$, there are $(2\mu+1)$ possible values to consider and there are n entries in vector Π, a lower bound for the complexity of Procedure 5.1 is

$\Theta((2\mu+1)^n)$. Usually, candidate Π where the absolute values of π_i, $i = 1, \cdots, n$, are small, e.g., $\Pi = [1, 1, 1]$ for the matrix multiplication algorithm, does not contribute a conflict-free mapping. So more sophisticated methods of finding the solution of Problem 2.2 may be possible. In general, together with Theorem 2.1, these necessary and sufficient conditions described in Theorems 3.1, 4.5, 4.6, and 4.7, depending on the dimension of the mapping matrix T, should be used to guide the solution search which is under investigation. In the following, the integer linear programming approach is discussed.

For the mapping matrix $T \in Z^{(n-1)\times n}$, Problem 2.2 can be formulated as an integer programming problem as follows.

$$\min f = \sum_{i=1}^{n} |\pi_i| \mu_i \tag{5.1}$$

$$\text{subject to } \begin{cases} 1)\ \Pi D > \vec{0} \\ 2)\ SD = PK \text{ and} \\ \quad \sum_{j=1}^{r} k_{ji} \le \Pi \vec{d}_i, \qquad i = 1, \cdots, m \\ 3)\ \exists i \in \{1, \cdots, n\}, |f_i(\pi_1, \cdots, \pi_n)| > \mu_i \\ 4)\ \Pi \in Z^{1\times n} \end{cases} \tag{5.2}$$

where $T = \begin{bmatrix} S \\ \Pi \end{bmatrix}$, S, and P are given and f_i, $i = 1, \cdots, n$ are as defined in (3.2). The last two constraints required by Definition 2.2 are included implicitly in (5.2) because by Theorems 2.2 and 3.1 they are implied by constraint 3 in (5.2). Also, constraint 2 is not required if a new processor array is designed specially for the algorithm. By Proposition 3.2, constraint 3 in (5.2) is linear. So the formulation in (5.1) and (5.2) is an integer linear programming problem if, as in Examples 5.1 and 5.2, constraint 1 in (5.2) requires that $\pi_i > 0$, $i = 1, \cdots, n$. Actually, this integer linear programming problem can be further converted to a linear programming problem for some applications as indicated by Examples 5.1 and 5.2. This is because in these applications, the objective function is linear, and all extreme points of the solution sets are integral.

One more constraint $\gcd(f_1, \cdots f_n) = 1$, where f_i, $i = 1, \cdots, n$ are as defined in (3.2), should be added to the formulation described by (5.1) and (5.2) to guarantee the greatest common divisor of the resulting conflict vector is unity. However, this makes the problem more difficult to solve. Hence, this constraint is ignored and the feasibility of the conflict vector of the resulting solution is checked. In other words, the conflict vector may not feasible after the common factor of its entries is removed as indicated in Example 5.1. If all extreme points which have the minimum value of the objective function in (5.1) are not feasible, i.e., they do not contribute conflict-free mappings, then some heuristic analysis has to be used to find the optimal solution. In the Appendix, Example 5.1 illustrates this situation.

The general integer programming problem is NP-complete [29, p. 245]. However, for each fixed natural number n, there exists a polynomial time algorithm which solves the integer linear programming problem [29, p. 259]. This algorithm is a polynomial function of the number of constraints of the integer linear programming problem and a logarithmic function of the problem size variables μ_i, $i = 1, \cdots, n$. Many existing standard algorithms can be used to solve the formulation in (5.1) and (5.2) efficiently because in practice, the number of constraints and the algorithm dimension n are not large.

Example 5.1: Consider the matrix multiplication algorithm in Example 3.1 where the space mapping matrix is given as $S = [1, 1, -1]$ [23]. The dependence matrix D and index set J are shown in (3.4). To satisfy condition 1 in (5.2), each entry of the linear schedule vector Π must be positive, i.e., $\pi_i \ge 1$, $i = 1, \cdots, 3$. Therefore, the problem of finding an optimal linear schedule vector for the matrix multiplication algorithm is formulated as an integer linear programming problem:

$$\min f = \mu(\pi_1 + \pi_2 + \pi_3) \tag{5.3}$$

$$\text{subject to } \begin{cases} 1)\ \pi_i \ge 1, i = 1, 2, 3 \\ 2)\ SD = PK \text{ and} \\ \quad \sum_{j=1}^{r} k_{ji} \le \Pi \vec{d}_i, \qquad i = 1, \cdots, 3 \\ 3)\ \pi_2 + \pi_3 \ge \mu + 1, \text{ or } \pi_1 + \pi_3 \ge \mu + 1, \text{ or} \\ \quad |\pi_1 - \pi_2| \ge \mu + 1 \\ 4)\ \Pi \in Z^{1\times 3} \end{cases}$$

where the inequalities in constraint 3 are derived in Example 3.1 and shown in (3.5). As indicated in the Appendix, this problem can be converted to four linear programming problems. In the Appendix, techniques for solving linear programming problems are used to find the optimal solution. There are at least three potential optimal solutions $\Pi_2 = [1, \mu, 1]$ and $\Pi_3 = [\mu, 1, 1]$, $\Pi_{11} = [1, 2, \mu - 1]$. Their corresponding conflict vectors are $\bar{\gamma}_2 = [\mu + 1, -2, \mu - 1]^T$, $\bar{\gamma}_3 = [2, -(\mu + 1), 1 - \mu]^T$, and $\bar{\gamma}_{11} = [(\mu + 1), -\mu, 1]^T$, respectively. When μ is an odd number, Π_2 and Π_3 are not feasible because their conflict vectors are not feasible. So, for the cases where μ is an odd number, Π_{11} can be chosen as the optimal solution and for the cases where μ is an even number, all the three linear schedule vectors can be the optimal solution. The derivation of the solution is shown in the Appendix. Let us choose $\Pi^o = \Pi_{11} = [1, 2, \mu - 1]$. The total execution time is $t = \mu(2 + \mu) + 1$ according to (2.7). The matrix of interconnection primitives is $P = S = [1, 1, -1]$ and $K = I$, the identity matrix. Fig. 2 shows the block diagram of the linear array for multiplying two 4×4 matrices ($\mu = 3$). Fig. 3 shows the execution of the matrix multiplication algorithm by mapping matrix $T = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & 2 \end{bmatrix}$. Computation $c_{j1,j2} = c_{j1,j2} + a_{j1,j3} \cdot b_{j3,j2}$ indexed by $\vec{j} = [j_1, j_2, j_3]^T$ is executed at processor $S\vec{j}$ and at time $\Pi\vec{j}$. By inspecting Fig. 3, there are no computational conflicts. Also, as shown in Fig. 3 and explained in the Appendix, there is no link collision either which means no two data use the same link at the same time. As shown in Fig. 2, three data links are used, one for data A traveling from left to right, one for data B traveling from left to right, and one for data C traveling from right to left. Two buffers are needed for each PE. One is on the link for data A, or for dependence vector $\vec{d}_2$. The other is on the link for data C, or for dependence vector $\vec{d}_3$. □

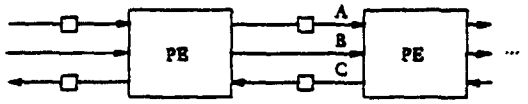


Fig. 2. Block diagram of the linear array for the matrix multiplication algorithm.

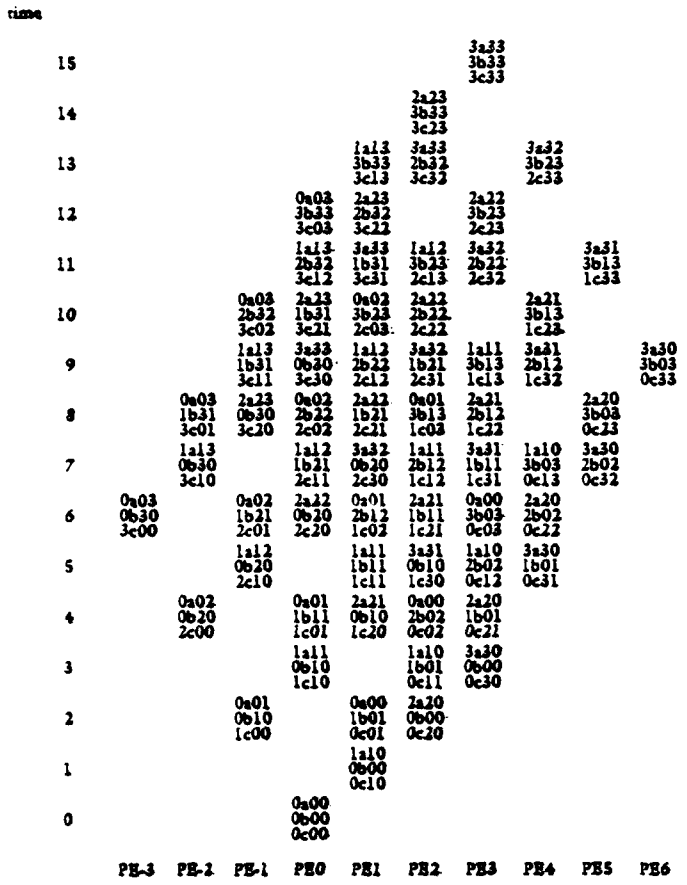


Fig. 3. The execution of multiplying two 4×4 matrices. The small block with the leftmost column being $[j_1, j_2, j_3]^T$ corresponds to the computation $c_{j_1 j_2} = c_{j_1 j_2} + a_{j_1 j_3} \cdot b_{j_3 j_2}$ which is executed at processor $j_1 + j_2 - j_3$ and at time $j_1 + 2j_2 + 2j_3$.

In [23], with the same space mapping matrix S, linear schedule vector $\Pi' = [2, 1, \mu]$ which is not optimal is used and the corresponding conflict vector is $\bar{\gamma} = [(\mu+1), -2-\mu, -1]^T$. The total execution time by Π' is $t' = \mu(3 + \mu) + 1$ which is longer than the optimal linear schedule Π°. Also, the number of buffers in [23] is $\sum_{i=1}^{3}(\Pi' \bar{d}_i - 1) = 3$ when $\mu = 3$. The systolic array designed in this paper only needs two buffers.

Example 5.2: Consider the transitive closure algorithm in Example 3.2 where the space mapping matrix is given as $S = [0, 0, 1]$ [22]. The dependence matrix D and index set J are shown in (3.6). To satisfy condition 1 in (5.2), it must have $\pi_2 > 0$, $\pi_3 > 0$, and $\pi_1 - \pi_2 > 0$, or $\pi_1 > \pi_2 > 0$. This means each entry of the linear schedule vector Π must be positive. Therefore, the problem of finding an optimal linear schedule vector for the transitive closure algorithm is formulated as an integer linear programming problem:

$$\min f = \pi_1(\mu + \pi_2 + \pi_3) \qquad (5.4)$$

$$\text{subject to} \begin{cases} 1)\ \pi_i \geq 1, i = 2, 3, \pi_1 - \pi_2 - \pi_3 \geq 1, \\ \quad \pi_1 - \pi_2 \geq 1, \pi_1 - \pi_3 \geq 1 \\ 2)\ SD = PK \text{ and} \\ \quad \sum_{j=1}^{r} k_{ji} \leq \Pi \bar{d}_i, \qquad i = 1, \cdots, 3 \\ 3)\ \pi_2 \geq \mu + 1, \text{or} \pi_1 \geq \mu + 1 \\ 4)\ \Pi \in Z^{1 \times 3} \end{cases}$$

where constraint 3 is derived in Example 3.2 and shown in (3.7). Again, as indicated in the Appendix, this problem can be formulated as two linear programming problems. The optimal solution of this algorithm is $\Pi^\circ = [\mu + 1, 1, 1]$ when $\mu \geq 2$. The derivation of the solution is shown in the Appendix. The total execution time is $t = \mu(3 + \mu) + 1$ according to (2.7). As mentioned in Section I, the solution found by the heuristic procedure in [22] is $\Pi' = [2\mu + 1, 1, 1]$ (notice that the lower and upper bounds for index points are 1 and n in [22] and therefore, $\mu = n - 1$ in [22]) and the total execution time is $t' = \mu(2\mu + 3) + 1$ which is much longer than for Π°. The matrix of interconnection primitives is $P = SD = [1, 0, -1, 0, -1]$ and $K = I$, the identity matrix. Clearly, there are no computational conflicts because the conflict vector of the mapping matrix $\begin{bmatrix} 0 & 0 & 1 \\ \mu+1 & 1 & 1 \end{bmatrix}$ is $\bar{\gamma} = [1, -(\mu + 1), 0]^T$ which is feasible according to Theorem 2.2. Also, as explained in the Appendix, there is no link collision either which means no two data use the same link at the same time. □

VI. Conclusions and Future Work

The first contribution of this paper is the use of the Hermite normal form of the mapping matrix by which all conflict vectors are expressed as integer combinations of a set of vectors from the nullspace of the mapping matrix. Based on the Hermite normal form, necessary and sufficient conditions are derived for computational conflict-free mappings. The second is the method of finding optimal time mappings of n-dimensional algorithms into $(n-2)$-dimensional processor arrays. These techniques can be used to map algorithms with n nested loops into linear or two-dimensional processor arrays with the total execution time minimized; they are especially useful for programming bit level processor arrays such that the total execution time is minimized.

Future work includes optimal solutions of the general mappings where $T \in Z^{k \times n}$ for arbitrary k, consideration of the number of buffers and length of wires required by the mappings, and investigation of the following two problems.

Problem 6.1 (Space-optimal and conflict-free mapping problem): Given an n-dimensional uniform dependence algorithm and a linear schedule vector, find a space mapping matrix $S \in Z^{(k-1) \times n}$ such that $T = \begin{bmatrix} S \\ \Pi \end{bmatrix}$ is conflict-free and the number of processors plus the wire length of the array is minimized.

Problem 6.2 (Optimal conflict-free mapping problem): Given an n-dimensional uniform dependence algorithm and a $(k-1)$-dimensional processor array, find a conflict-free mapping matrix $T \in Z^{k \times n}$ such that a certain criterion is optimized.

In general, in Problem 2.2, space mapping matrix S is given and usually is not a function of problem size variables μ_i, $i = 1, \cdots, n$; in Problem 6.1, linear schedule vector Π is given.

ossibly by the optimization procedure proposed in [16], and sually is not a function of problem size variables; and in roblem 6.2, both S and Π are not given and possibly are oth functions of problem size variables.

APPENDIX

Discussion of Example 5.1: Let us design a new linear ystolic array for the matrix multiplication algorithm. Thus, onstraint 2 in (5.3) can be ignored at this moment. For an nteger linear programming problem with convex solution set, f all of its extreme points are integral, then one of the extreme oints is the optimal solution of that problem [29, p. 232]. Now he solution set of the integer programming problem in (5.3) is ot convex because of constraint 3 although all extreme points re integral. One way to solve this problem is to partition the olution set as four convex subsets and then to find all optimal olutions for all the disjoint solution subsets. If the one with he smallest value of the objective function is satisfactory, then t is the optimal solution of the integer programming problem n (5.3).

Now let us partition the solution set of the integer programming problem in (5.3) as four subsets which are expressed as ollows.

$$
\begin{aligned}
&(I)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3)\\
&\text{subject to} \begin{cases} 1)\ \pi_i \geq 1, i = 1,2,3\\ 2)\ \pi_2 - \pi_3 \geq \mu + 1\\ 3)\ \pi_1 - \pi_3 \leq \mu + 1\\ 4)\ \pi_1 - \pi_2 \leq \mu + 1\\ 5)\ -\pi_1 + \pi_2 \leq \mu + 1\\ 6)\ \Pi \in Z^{1\times 3} \end{cases}\\
&(II)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3)\\
&\text{subject to} \begin{cases} 1)\ \pi_i \geq 1, i = 1,2,3\\ 2)\ \pi_1 + \pi_3 \geq \mu + 1\\ 3)\ \Pi \in Z^{1\times 3} \end{cases}\\
&(III)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3)\\
&\text{subject to} \begin{cases} 1)\ \pi_i \geq 1, i = 1,2,3\\ 2)\ \pi_1 - \pi_2 \geq \mu + 1\\ 3)\ \Pi \in Z^{1\times 3} \end{cases}\\
&(IV)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3)\\
&\text{subject to} \begin{cases} 1)\ \pi_i \geq 1, i = 1,2,3\\ 2)\ -\pi_1 + \pi_2 \geq \mu + 1\\ 3)\ \Pi \in Z^{1\times 3}. \end{cases}
\end{aligned}
\tag{a.1}
$$

Each of the above problems is an integer linear programming problem with a convex solution set. It can be checked that every extreme point of these convex sets is integral. Each extreme point is the solution of three of the following seven equations: $\pi_1 = 1$, $\pi_2 = 1$, $\pi_3 = 1$, $\pi_2+\pi_3 = \mu+1$, $\pi_1+\pi_3 = \mu+1$, $\pi_1-\pi_2 = \mu+1$, and $\pi_2-\pi_1 = \mu+1$. There are ten such solutions from these seven equations which satisfy $\Pi D > \bar{0}$ as follows. $\Pi_1 = [1, 1, \mu]$, $\Pi_2 = [1, \mu, 1]$, $\Pi_3 = [\mu, 1, 1]$, $\Pi_4 = [1, \mu - 2, 1]$, $\Pi_5 = [\mu + 2, 1, 1]$, $\Pi_6 = [1, \mu + 2, \mu]$, $\Pi_7 = [\mu - 2, 1, \mu]$, $\Pi_8 = [\mu, \mu, 1]$, $\Pi_9 = [2\mu + 1, \mu, 1]$, and $\Pi_{10} = [\mu, 2\mu + 1, 1]$. Extreme points with the shortest execution time are Π_1, Π_2, and Π_3. Π_1 is not feasible because the corresponding conflict vector $[1, -1, 0]^T$ is not feasible. Conflict vectors for Π_2 and Π_3 are $[(\mu + 1), -2, (\mu - 1)]^T$ and $[2, -(\mu + 1), (1 - \mu)]^T$, respectively. When μ is an even number, Π_2 and Π_3 are feasible. However, when μ is odd, both Π_2 and Π_3 are not feasible. The reason why some vectors in the solution space are not feasible is as follows. In (5.3) the constraint $\gcd(f_1, \cdots, f_n) = 1$, where f_i, $i = 1, \cdots, n$ are as defined in (3.2), is not included because this constraint makes the problem more difficult to solve. When all extreme points with the shortest execution time do not contribute conflict-free mappings for a particular μ, other solutions which are not extreme points and have the shortest execution time should be considered. However, those nonfeasible extreme points provide a lower bound on the execution time by all feasible solutions. For example, when μ is an odd number, all the three extreme points with the shortest execution time Π_1, Π_2, and Π_3 do not contribute conflict-free mappings. Because the sum of the entries of each of Π_1, Π_2, and Π_3 are $\mu + 2$, other solutions with the same execution time have the property $\pi_1 + \pi_2 + \pi_3 = 2 + \mu$. Let us consider $\Pi_{11} = [1, 2, \mu - 1]$, a solution of equations $\pi_1 = 1$ and $\pi_1 + \pi_2 + \pi_3 = \mu + 2$. The conflict vector of Π_{11} is $\bar{\gamma}_{11} = [\mu + 1, -\mu, 1]^T$. Thus, Π_{11} is feasible and optimal because the conflict vector is feasible and it has the same execution time as the extreme points Π_1, Π_2, and Π_3 which have the shortest execution time. Therefore, for the matrix multiplication algorithm, when μ is odd, one optimal solution is Π_{11} and when μ is even, Π_2, Π_3, and Π_{11} could be chosen as the optimal solution.

Let us design the linear systolic array for mapping matrix $T = \begin{bmatrix} S \\ \Pi_{11} \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & 2 \end{bmatrix}$ where $\mu = 3$. If $P = [1, 1, -1]$ is chosen as the matrix of interconnection primitives and $K = I$ (the identity matrix), then $SD = PK$, $\sum_{j=1}^{3} k_{j1} = 1 = \Pi\bar{d}_1 = 1$, $\sum_{j=1}^{3} k_{j2} = 1 \leq \Pi\bar{d}_2 = 2$, $\sum_{j=1}^{3} k_{j3} = 1 \leq \Pi\bar{d}_3 = 2$ and therefore, constraint 2 in (5.3) is satisfied. Because $\Pi\bar{d}_2 - \sum_{j=1}^{3} k_{j2} + \Pi\bar{d}_3 - \sum_{j=1}^{3} k_{j3} = 2$, two buffers are needed, one for the link of data A induced by dependence $\bar{d}_2$ and the other for the link of data C induced by dependence $\bar{d}_3$. The systolic structure and the execution are shown in Figs. 2 and 3, respectively.

Notice that there is no data link collision because in every column and every row of matrix K there is only one nonzero entry $k_{ii} = 1$, $i = 1, \cdots, 3$. This means that when data pass from the source to the destination, they use the data link just once (one hop between two PE's). Data link collisions occur only if data use links more than once when passing from the source to the destination. For example, if the space mapping $S' = [1, 1, \mu]$ and $P' = [1, 1, 1]$, then to satisfy the condition $SD = PK$, one possible set of values for K is $k_{11} = k_{22} = 1$, $k_{33} = \mu$ and $k_{ij} = 0$, $i \neq j$. Thus, the distance between the source and destination for data C is μ PE's and data C will take μ hops over the third link in P, or the link for C to reach the destination. Suppose PE_i, $i = 1, \cdots, \mu$, are sending data x_{ij} (corresponding to c_{ij} of matrix C) to $PE_{i+\mu}$ at time t_j, $j = 1, \cdots, \mu$. Then at time t_1, x_{11} is on the link between PE_1 and PE_2. At time t_2, two pieces of data x_{11} and x_{22} are on the link between PE_2 and PE_3 and so on. At time $t_\mu - 1$, $\mu - 1$ pieces of data $x_{11}, x_{22}, \cdots, x_{\mu-1,\mu-1}$

are on the link between $PE_{\mu-1}$ and PE_{μ}. So, link collisions exist after time t_1. This is caused by $k_{33} = \mu$.

Discussion of Example 5.2: Similarly, the integer programming problem in (5.4) can be converted to the following two integer linear programming problems.

$$\text{subject to } \begin{cases} (I)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3) \\ 1)\ \pi_i \geq 1, i = 2,3, \pi_1 - \pi_2 - \pi_3 \geq 1, \\ \quad \pi_1 - \pi_2 \geq 1, \pi_1 - \pi_3 \geq 1 \\ 2)\ \pi_2 \geq \mu + 1 \\ 3)\ \pi_1 \leq \mu + 1 \\ 4)\ \Pi \in Z^{1\times 3} \end{cases} \tag{a.2}$$

$$\text{subject to } \begin{cases} (II)\ \min f = \mu(\pi_1 + \pi_2 + \pi_3) \\ 1)\ \pi_i \geq 1, i = 2,3, \pi_1 - \pi_2 - \pi_3 \geq 1, \\ \quad \pi_1 - \pi_2 \geq 1, \pi_1 - \pi_3 \geq 1 \\ 2)\ \pi_1 \geq \mu + 1 \\ 3)\ \Pi \in Z^{1\times 3}. \end{cases}$$

Again, because all extreme points of the convex solution subsets are integral, techniques for linear programming can be applied, which check all extreme points of the solution subsets. One of the extreme points with the minimum value of the objective function f is the optimal one. Again, each extreme point is a solution of the following seven equations. $\pi_2 = 1$, $\pi_3 = 1$, $\pi_1 - \pi_2 - \pi_3 = 1$, $\pi_1 - \pi_2 = 1$, $\pi_1 - \pi_3 = 1$, $\pi_1 = \mu + 1$, and $\pi_1 = \mu + 1$. Four such extreme points are $\Pi_1 = [\mu + 1, 1, 1]$, $\Pi_2 = [\mu + 1, 1, \mu - 1]$, $\Pi_3 = [\mu + 1, 1, \mu]$, and $\Pi_4 = [\mu + 1, \mu - 1, 1]$. When $\mu \geq 2$, all the above linear schedule vectors satisfy constraint $\Pi D > \bar{0}$. The corresponding conflict vectors, according to (3.2), are $\bar{\gamma}_1 = [1, -(\mu + 1), 0]^T$, $\bar{\gamma}_2 = [1, -(\mu + 1), 0]^T$, $\bar{\gamma}_3 = [1, -(\mu+1), 0]^T$, and $\bar{\gamma}_4 = [\mu-1, -(\mu+1), 0]^T$. Vectors $\bar{\gamma}_i$, $i = 1, \cdots, 3$, are feasible and $\bar{\gamma}_4$ is feasible if μ is an even number. The extreme point with the minimum value of the objective function f is $\Pi^o = \Pi_1 = [\mu + 1, 1, 1]$ and the total execution time by Π_1 is $\mu(\mu + 3) + 1$ according to (2.7).

If the matrix of interconnection primitives $P = SD = [1, 0, -1, 0, -1]$ is chosen and $K = I$, the identity matrix, then constraint 2 in (5.4) is satisfied. Similar to the design for the matrix multiplication algorithm, there is no data link collision because in every column and every row of matrix K there is only one nonzero entry $k_{ii} = 1$, $i = 1, \cdots, 3$.

ACKNOWLEDGMENT

We would like to thank one referee for the ten-page constructive and very helpful comments and Y. X. Wang for drawing the figures in this paper.

REFERENCES

[1] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computation for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, July 1967.

[2] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. C-35, pp. 1–12, Jan. 1986.

[3] P. R. Cappello and K. Steiglitz, "Unifying VLSI array designs with geometric transformations," in *Proc. 1983 Int. Conf. Parallel Processing*, pp. 448–457.

[4] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proc. 11th Annu. Symp. Comput. Architecture*, 1984, pp. 208–214.

[5] S. K. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Stanford Univ., Stanford, CA, Oct. 1985.

[6] M. Chen, "A design methodology for synthesizing parallel algorithms and architectures," *J. Parallel Distributed Comput.*, Dec. 1986, pp. 461–491.

[7] J.-M. Delosme and I. C. F. Ipsen, "An illustration of a methodology for the construction of efficient systolic architectures in VLSI," in *Proc. Second Int. Symp. VLSI Technol., Syst. Appl.*, 1985, pp. 268–273.

[8] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[9] C. Guerra and R. Melhem, "Synthesizing non-uniform systolic designs," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 765–771.

[10] G.-J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 66–77, Jan. 1985.

[11] M. T. O'Keefe and J. A. B. Fortes, "A comparative study of two systematic design methodologies for systolic arrays," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 672–675.

[12] J. A. B. Fortes and F. Parisi-Presicce "Optimal linear schedule for the parallel execution of algorithms," in *Proc. 1984 Int. Conf. Parallel Processing*, pp. 322–328.

[13] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[14] J.-K. Peir and R. Cytron, "Minimum distance: A method for partitioning recurrences for multiprocessors," in *Proc. 1987 Int. Conf. Parallel Processing*, pp. 217–225.

[15] V. P. Roychowdhury and T. Kailath, "Subspace scheduling and parallel implementation of non-systolic regular iterative algorithms," *J. VLSI Signal Processing*, vol. 1, 1989.

[16] W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," *IEEE Trans. Comput.*, vol. 40, pp. 723–742, June 1991.

[17] S. Y. Kung, S. C. Lo, and P. S. Lewis, "Optimal systolic design for the transitive closure and the shortest path problems," *IEEE Trans. Comput.*, vol. C-36, pp. 603–614, May 1987.

[18] G. Strang, *Linear Algebra and its Applications*, second ed. New York: Academic, 1980.

[19] R. Cytron, "Doacross: Beyond vectorization for multiprocessors (extended abstract)," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 836–844.

[20] R. Kannan and A. Bachem, "Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix," *SIAM J. Comput.*, vol. 8, no. 4, pp. 499–507, Nov. 1979.

[21] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI array processors," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980, sect. 8.3.

[22] P. Lee and Z. M. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays," *IEEE Trans. Parallel Distributed Syst.*, vol. 1, pp. 64–76, Jan. 1990.

[23] ——, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. Comput.*, vol. 37, pp. 1578–1598, Dec. 1988.

[24] D. A. Padua, "Multiprocessors: Discussion of theoretical and practical problems," Ph.D dissertation, Univ. of Illinois at Urbana–Champaign, Rep. UIUCDCS-R-79-990, Nov. 1979.

[25] Y. Wong and J.-M. Delosme, "Optimal systolic implementation of N-dimensional recurrences," in *IEEE Proc. ICCD*, 1985, pp. 618–621.

[26] V. E. Taylor and J. A. B. Fortes, "Using RAB to map algorithms into bit-level systolic arrays," in *Proc. Int. Conf. Supercomput.*, May 1987.

[27] L. J. Mordell, *Diophantine Equations*. New York: Academic, 1969, p. 30.

[28] M. T. O'Keefe and J. A. B. Fortes, "Bit level processor array: Current architectures and a design and a programming tool," in *Proc. 1988 Int. Symp. Circuit Syst.*, Helsinki, Finland, June 1988, pp. 2751–2755.

[29] A. Schrijver, *Theory of Linear and Integer Programming*. New York: Wiley, 1986.

[30] W. Shang, "Scheduling, partitioning and mapping of uniform dependence algorithms on processor arrays," Ph.D. dissertation, Purdue Univ., W. Lafayette, IN 47907, May 1990.

[31] T. Blank, "The MasPar MP-1 Architecture," in *Proc. 35th IEEE Comput. Soc. Int. Conf.—Spring Compcon 90*, San Francisco, CA, Feb. 1990.

[32] D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, vol. 71, pp. 113–120, Jan. 1983.

[33] R. Davis and D. Thomas, "Systolic array chip matches the pace of high-speed processing," *Electron. Design*, Oct. 31, 1984.

[34] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*. Bristol, Adam Hilger: 1981.

pp. 178–192.
[35] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 377–384.
[36] J. A. B. Fortes and B. W. Wah, "Systolic array—From concept to implementation," *IEEE Comput. Mag.*, pp. 12–17, July 1987.
[37] Z. Yang, W. Shang, and J. A. B. Fortes, "One-to-one time mappings of nested algorithms into lower dimensional processor arrays," in *Proc. Sixth IEEE Int. Parallel Processing Symp.*, Mar. 1992, Beverly Hills, CA, pp. 156–164.

Weijia Shang (S'88–M'90) received the B.S. degree in electrical and computer engineering from Changsha Institute of Technology, China, in 1982 and the M.S. and Ph.D. degrees in electrical engineering from Purdue University, West Lafayette, IN, in 1984 and 1990, respectively.

She is currently an Assistant Professor in the Center for Advanced Computer Studies, the University of Southwestern Louisiana, Lafayette. Her research interests include parallel processing, computer architecture, algorithm transformation, processor array programming, special purpose VLSI bit-level processor array design, and optimizing compiler technique.

Dr. Shang is a member of the Association for Computing Machinery.

Jose A. B. Fortes (S'80–M'83) received the Licenciatura em Engenharia Electrotecnica from the Universidade de Angola in 1978, the M.S. degree in electrical engineering from the Colorado State University, Fort Collins, in 1981, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1984.

In 1984, he joined the faculty of the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he currently is an Associate Professor. From July 1989 to July 1990 he served at the National Science Foundation as program director for microelectronics systems architecture. His research interests are in the areas of parallel processing, fault-tolerant computing, and VLSI computer architecture on which he coauthored over 50 technical papers. His research has been funded by the Office of Naval Research, AT&T Foundation, General Electric, and the National Science Foundation.

Dr. Fortes is a member of the IEEE Professional Society. He is on the Editorial Boards of the *Journal of Parallel and Distributed Computing* and the *Journal of VLSI Signal Processing*.

REFERENCE NO. 7

Chean, M., and Fortes, J. A. B., "The Full-Use-of-Suitable-Spaces (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerance Processor Arrays," *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 564-571.

Note - This paper describes a new technique for processor array reconfiguration and discusses its performance. It allows a processor array to reconfigure in the presence of up to as many faults as the number of spare processors available with a probability of more than 98%.

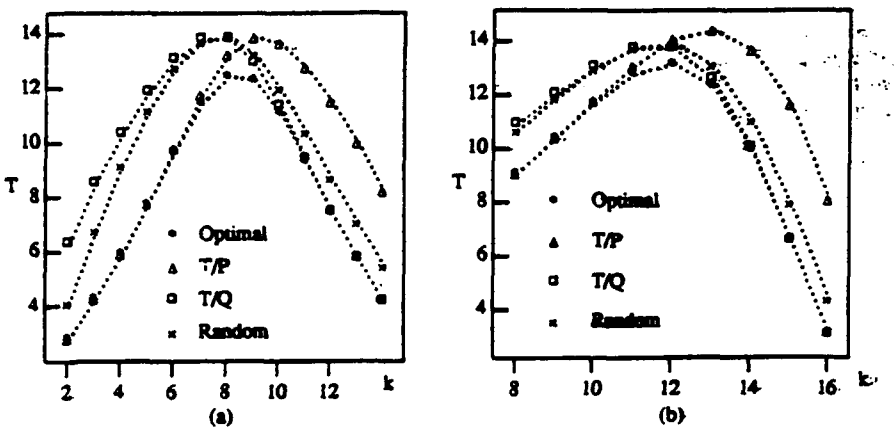


Fig. 6. The expected diagnosis time of two k-out-of-16 structures. (a) A k-out-of-16 system (p_1: 0.3–0.7). (b) A k-out-of-16 system (p_1: 0.71–0.99).

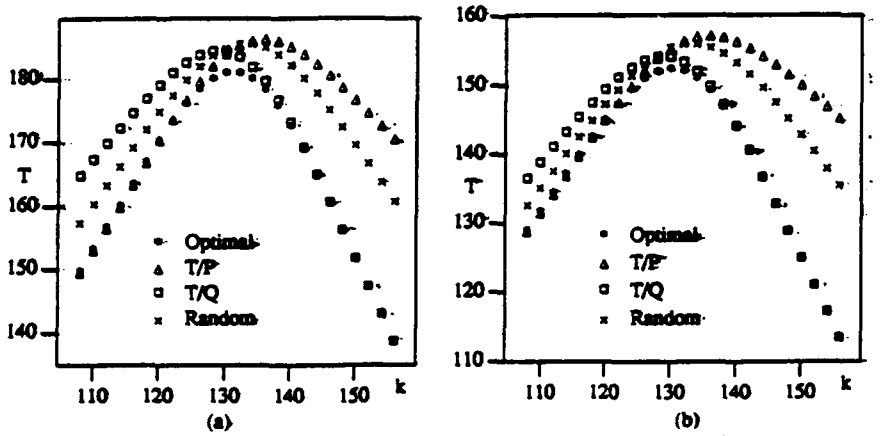


Fig. 7. k-out-of 256 structures with different T_f, p_1: (0.3, 0.7). (a) $T_s = 1$, $T_f = 0.5$. (b) $T_s = 1$, $T_f = 0.25$.

vidual units and the expected time for testing each unit is known. A proof of the optimality of a diagnosis algorithm was given for k-out-of-n structures along with a method of compactly representing an optimal solution. Simulation results illustrate the reduction in diagnosis time achieved by exploiting knowledge of the repair strategy as well as knowledge of the expected yield and test time of each unit in the structure.

References

[1] M.-F. Chang, W. K. Fuchs, and J. H. Patel, "Diagnosis and repair of memory with coupling faults," *IEEE Trans. Comput.*, vol. 38, pp. 493–500, Apr. 1989.

[2] R. W. Haddad and A. T. Dahbura, "Increased throughput for the testing and repair of RAMs with redundancy," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, 1987, pp. 230–233.

[3] W. K. Huang and F. Lombardi, "Minimizing the cost of repairing WSI memories," in *Proc. IEEE Int. Conf. Wafer Scale Integration*, Jan. 1989, pp. 183–192.

[4] A. V. Ferris-Prabhu, L. D. Smith, H. A. Bonges, and J. K. Paulsen, "Radial yield variations in semiconductor wafers," *IEEE Circuits Devices*, pp. 42–47, Mar. 1989.

[5] A. B. Leslie and A. F. Matta, "Wafer-level testing with a membrane probe," *IEEE Design Test Comput.*, vol. 6, pp. 10–17, Feb. 1989.

[6] J. Halpern, "Fault testing for a k-out-of-n system," *Oper. Res.*, vol. 22, pp. 1267–1271, Nov.–Dec. 1974.

[7] S. Salloum and M. A. Breuer, "An optimum testing algorithm for some symmetric coherent systems," *J. Math. Anal. Appl.*, vol. 101, pp. 170–194, 1984.

[8] Y. Ben-Dov, "Optimal testing procedures for special structures of coherent systems," *Management Sci.*, vol. 27, pp. 1410–1420, Dec. 1981.

[9] ——, "A branch and bound algorithm for minimizing the expected cost of testing coherent systems," *Euro. J. Oper. Res.*, vol. 7, pp. 284–289, 1981.

[10] P. Jedrzejowicz, "Minimizing the average cost of testing coherent systems: Complexity and approximation algorithms," *IEEE Trans. Reliability*, vol. R-32, pp. 66–70, Apr. 1983.

[11] R. Butterworth, "Some reliability fault-testing models," *Oper. Res.*, vol. 20, pp. 335–343, Mar.–Apr. 1972.

[12] A. Tucker, *Applied Combinatorics*. New York: Wiley, 1984, p. 447.

The Full-Use-of-Suitable-Spares (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerant Processor Arrays

MENGLY CHEAN AND JOSÉ A. B. FORTES

Abstract—A general approach to hardware reconfiguration is proposed for VLSI/WSI fault-tolerant processor arrays. The technique, called FUSS (full use of suitable spares), uses an indicator vector, the

Manuscript received July 2, 1989; revised November 30, 1989. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contracts 00014-85-k-0388 and 00014-88-k-0723.

M. Chean is with Bellaire Research Center, Shell Development Company, Houston, TX 77025.

J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 8933877.

surplus vector, to guide the replacement of faulty processors within an array. Analytical study of the general FUSS algorithm shows that there is a linear relationship between the array size and the area of interconnect required for reconfiguration to be 100% successful.

In an instance of FUSS, called *simple FUSS*, reconfiguration is done by simply "shifting" up or down faulty processors along their corresponding columns according to the surplus vector's entries. The surplus vector is progressively updated after each column is reconfigured. The reconfiguration is successful when the surplus vector becomes null. Simple FUSS is discussed in detail and evaluated. Simulations show that when the number of faulty processors is equal to that of spare processors, simple FUSS can achieve a probability of survival as high as 99%; this is far better than the probability of survival of existing schemes with similar complexity in execution time and interconnection hardware.

Index Terms— Fault-tolerant, reconfiguration, redundancy, reliability, survivability.

I. Introduction

A difficult problem faced in the design of fault-tolerant processor arrays is how to reconfigure hardware in order to dynamically replace faulty processors with functional spares. This paper presents a general and ideal approach, as well as a practical instance of it, that guarantees successful reconfiguration with higher probability than presently known approaches of comparable space-time complexity. For the proposed scheme, called FUSS (full use of suitable spares), basic reconfiguration and interconnection requirements, and probability of survival are presented and discussed.

Many reconfiguration schemes (see [1] for a recent discussion and classification of extant approaches) fail to use advanced knowledge of the distribution of faulty processors in the array. As the algorithm proceeds, faulty processors are replaced locally. Thus, spare processors that have not been used may not be available for replacing faulty processors that the algorithm encounters later. This results in low probability of survival (the conditional probability that the reconfiguration is successful given that the array has a number of faulty processors), especially when the number of faulty processors approaches the number of spares. FUSS uses an indicator vector, the *surplus vector*, to guide the replacement of faulty processors within an array. This vector has as many entries as the number of rows of the array. The initial value of entry i is the sum of spares in the subarray consisting of rows 1 through i, minus the sum of faulty processors in the subarray. Based on the entries of the surplus vector, reconfiguration is accomplished by "shifting" faulty processors from regions having the most faulty processors to regions having fewer faulty processors. The surplus vector is progressively updated during shifting, and reconfiguration is successful when the surplus vector becomes null.

General FUSS has, in its most general and ideal case, 100% probability of survival. The general FUSS algorithm is studied (Section II), and then a practical instance of FUSS, called *simple FUSS*, is discussed in detail and evaluated (Section III). Simulations show that simple FUSS can achieve a probability of survival as high as 99% at the maximum number of faults allowable in an array (i.e., when there are as many faulty processors as there are spares); this is far better than the probability of survival of existing schemes with similar complexity in execution time and redundant hardware (see [2], [5], [6], and references therein).

II. FUSS Approach to Array Reconfiguration

A. Definitions and Notations

In FUSS, the array is assumed to be $M \times (N + C)$, where M is the total number of rows, N is the total number of columns, and C is the total number of spare columns. Fig. 1(a) shows such an array.

A *fault-tolerant processor array* (FTPA) is a two-dimensional array of identical and regularly interconnected processing elements in which redundant circuitry and spares are incorporated. A *physical FTPA* is a nonreconfigured FTPA which may contain faulty elements. A *logical FTPA*, on the other hand, is a reconfigured FTPA that is fully operational; faulty physical processors have been replaced by operational spares.

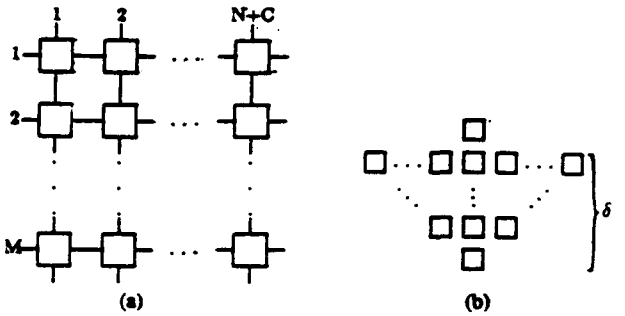


Fig. 1. FUSS FTPA model. (a) FUSS FTPA. (b) FUSS lower support-domain Δ_l.

A *cell* is a processing element (PE) of an FTPA. The notation (i, j) refers to the cell with physical coordinates i, j. The cell with logical cordinates i, j is referred to as $[i, j]$. A *reconfiguration mapping* ϕ is a function that maps a logical cell to a physical cell, e.g., $\phi([i, j]) = (x, y)$. The inverse of ϕ maps a functional physical cell to a logical cell, e.g., $\phi^{-1}((i, j)) = [m, n]$. Also, $\phi_i(\cdot)$ and $\phi_j(\cdot)$ map a logical cell to the corresponding physical row and the corresponding physical column, respectively. For example, $\phi_i([m; n]) = x$ if logical cell $[m, n]$ is on physical row x. The functions $\phi_i^{-1}(\cdot)$ and $\phi_j^{-1}(\cdot)$ are defined similarly.

An *available* cell is a good, functional cell, whereas an *unavailable* cell refers to either a faulty cell or a fault-free cell that is already used to replace another cell. *Shift* refers to logical replacement of an unavailable cell by an available one. Cell (i, j) is shifted to cell (m, n) if (m, n) replaces (i, j).

A *status matrix* is an $M \times (N+C)$ matrix whose entries represent the status of cells of an array. In a physical FTPA, the status of a cell is either 0 (fault-free) or 1 (faulty). In a logical FTPA, the status of a cell may also include a higher number $(2, 3, \cdots)$ to indicate that it replaces another cell.

A *support-domain* Δ of a cell is a set of cells called *supporters*, each of which may replace the cell when it becomes unavailable. Two symmetrical support-domains are defined for each cell in the physical array. Fig. 1(b) shows the lower support-domain, denoted by Δ_l; the upper support-domain (Δ_u) is similar except that it is the mirror image of the one shown. A support-domain has a triangular shape (except near the array boundary) such that the Manhattan distance[1] between the cell and any of its farthest supporters is the same; this distance, denoted by δ, is referred to as *support radius*.

The *connection window* W of a cell is the set of all physical cells that can be logical neighbors to the cell. Two connection windows are associated with a cell, *horizontal window* W_H and *vertical window* W_V. The size of a window is the number of cells it contains.

The *probability of survival* or *survivability* σ is the conditional probability of success in reconfiguring an array given that the array has a number of faulty cells. The *demand-spare* ρ is the ratio of the number of faulty cells to the total number of spares in the array (i.e., the demand for spares normalized with respect to the number of spares).

Let $B = [b_{i,j}]$ be the status matrix of a physical FTPA; i.e., $b_{i,j} = 1$ if (i, j) is faulty, and $b_{i,j} = 0$ if (i, j) is fault-free. The following is a list of additional symbols used throughout the paper and their meaning.

- T An $M \times M$ lower triangular matrix such that $T = [t_{i,j}]$: $t_{i,j} = 0$ if $i < j$, and $t_{i,j} = 1$ otherwise.
- u Unit vector; $u = [1, 1, \cdots, 1]^T$
- n Integer vector; $n = [1, 2, \cdots, M]^T$.

[1] The distance between cells, say (i, j) and (k, l), is defined as the Manhattan distance between them, i.e.,

$$d((i, j), (k, l)) = |i - k| + |j - l|.$$

The unit is assumed to be the cell side, the cell being a square.

f Fault vector; $f = [f_1, f_2, \cdots, f_M]^T$, $f_i = \sum_{j=1}^{N+C} b_{i,j}$.

F Total number of faulty cells in the array; $F = \sum_{i=1}^{M} f_i$.

S Total number of spare cells in the array; $S = MC$. The spare-demand defined previously is given by $\rho = F/S$.

s Surplus vector; $s = [s_1, s_2, \cdots, s_M]^T$, where s_i represents the *accumulated surplus* in the subarray consisting of rows 1 through i (total number of spares in the subarray minus total number of faulty cells in the subarray); see (2.3) below.

All cells are assumed identical and self-testing. Faults are assumed independent and uniformly distributed in the physical array.

B. Basic Concepts

The fault vector and the surplus vector can be written, respectively,

$$f = Bu \tag{2.1}$$

$$s = Cn - Tf. \tag{2.2}$$

Each entry s_i in s indicates the accumulated surplus in the subarray consisting of rows 1 through i:

$$s_i = \sum_{j=1}^{i} (C - f_j) = Ci - \sum_{j=1}^{i} f_j. \tag{2.3}$$

a) If $s_i > 0$, then $C \cdot i > \sum_{j=1}^{i} (\sum_{k=1}^{N+C} b_{j,k})$. This means that the sum of spares in rows 1 through i is greater than the number of faulty cells in rows 1 through i; thus, row i has extra cells available for use by faulty cells in rows $i+1, i+2, \cdots, M$.

b) If $s_i < 0$, then $C \cdot i < \sum_{j=1}^{i} (\sum_{k=1}^{N+C} b_{j,k})$; row i has a "cell-deficit" and needs to use available cells from rows $i+1, i+2, \cdots, M$.

c) If $s_M < 0$, then the total number of spares is less than the total number of faults. In this case, the array is not reconfigurable and the algorithm must exit with a failure.

The basic strategy of the proposed reconfiguration scheme is to shift cells around according to these needs: for instance, unavailable cells are shifted to row i if $s_i > 0$, or out of row i if $s_i < 0$. The generation of fault vector and surplus vector is shown in Fig. 2, for a $5 \times (5+2)$ physical FTPA ($M = N = 5$, $C = 2$).

C. General FUSS Algorithm

The general FUSS algorithm accepts a physical FTPA as the input, generates a logical FTPA as its output, and can be divided into four parts which execute in the following order: 1) array preprocessing, 2) surplus normalization, 3) fault shifting, and 4) cell interconnection. This subdivision is done for clarity as well as for monitoring the algorithm progress. For instance, the reconfiguration may be successful when "fault shifting" is completed; however, it may fail when "cell interconnection" is applied if the provided interconnection network is restricted. For this reason, the study of the effect of limited interconnect on the algorithm can be easily done. A flowchart of the general FUSS algorithm is provided in Fig. 3.

Array Preprocessing: The array preprocessing consists of generating fault vector f and surplus vector s, and, based on the value of s_M, deciding to abort or continue the reconfiguration. It can be summarized as follows:

1) Generate f and s using (2.1) and (2.2).
2) If $s_M < 0$, exit with failure.

Surplus Normalization: If $s_M > 0$, s_M extra cells in rows M are available for use by "imaginary" faulty cells in nonexistent rows $M+1, M+2, \cdots$. Shifting these faulty cells to available cells in row M would be required by the rules used in the fault-shifting phase of FUSS. A better solution is to make $s_M = 0$, which is one of the objectives of the surplus normalization. A second reason applies to FTPA's with a number of faulty cells much smaller than the number of spares. For instance, if in a surplus vector there exists a set of positive entries with increasing values, each row of FTPA corresponding to each entry of this set must have a number of faulty cells less than the number of spare cells. This means that shifting faulty cells in these rows may not be needed. In general, unnecessary shifting may result from large accumulated surplus. Thus, surplus normalization is preferred to minimize the number of shiftings required by the actual faults.

$$f = Bu = \begin{bmatrix} 0&0&1&0&0&1&1 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 1&0&0&1&0&1&0 \\ 0&1&0&0&0&0&0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 0 \\ 3 \\ 1 \end{bmatrix}$$

$$s = Cn - Tf = 2 \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} - \begin{bmatrix} 1&0&0&0&0 \\ 1&1&0&0&0 \\ 1&1&1&0&0 \\ 1&1&1&1&0 \\ 1&1&1&1&1 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 0 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 3 \\ 2 \\ 3 \end{bmatrix}$$

Fig. 2. Example of generation of fault vector f and surplus vector s for a physical FTPA with status matrix B.

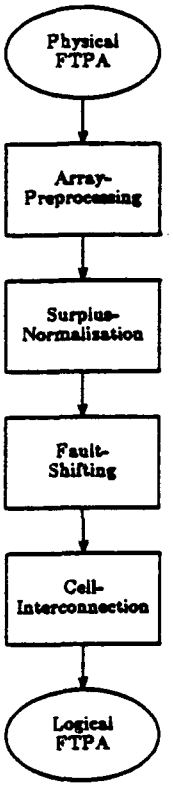


Fig. 3. General FUSS flowchart.

Surplus normalization is outlined below. The notation $S_{i/j}$ refers to an ordered list with elements $(s_i, s_{i+1}, \cdots, s_j)$, $i \le j$.

1) Given $s = [s_1, s_2, \cdots, s_M]^T$, $s_M > 0$, set $I = 1$.
2) Let $S_{I/M} = (s_I, s_{I+1}, \cdots, s_M)$
3) Partition $S_{I/M}$ into two disjoint lists:
   $S_{I/I+k} = (s_I, s_{I+1}, \cdots, s_{I+k})$, $I + k < M$ and $s_i \le 0$, $i = I, I+1, \cdots, I+k$;
   $S_{I+k+1/M} = (s_{I+k+1}, s_{I+k+2}, \cdots, s_M)$, $s_{I+k+1} > 0$.
4) Partition $S_{I+k+1/M}$ into two disjoint lists (one of which may be empty):
   $S_{I+k+1/l} = (s_{I+k+1}, s_{I+k+2}, \cdots, s_{l-1}, s_l)$, $l < M$, $s_{l-1} \le s_l$, $i = I+k+2, I+k+3, \cdots, l-1$, and $s_{l-1} > s_l$ ($S_{I+1+k/l-1}$ has elements with an increasing order of values);
   $S_{l+1/M} = (s_{l+1}, s_{l+2}, \cdots, s_M)$.
5) If $S_{I+k+1/l} = \phi$ (i.e., there are no two consecutive elements $s_l, s_{l+1} \in S_{I+k+1/l}$ such that $s_l > s_{l+1}$), then set $s_l \leftarrow 0$, $\forall s_l \in S_{I+k+1/M}$; go to 8).
6) Otherwise,

a) offset $\leftarrow \begin{cases} \min(s_l, s_M) & \text{if } s_l \geq 0 \\ 0 & \text{if } s_l < 0 \end{cases}$

b) $s_l \leftarrow s_l - \text{offset}$, $\forall s_l \in S_{l+k+1/M}$;

c) $\forall s_i \in S_{l+k+1/l}$ if $s_i < 0$ then reassign $s_i \leftarrow 0$

7) If $s_M > 0$, $l \leftarrow l + 1$ and go to 2).

8) Return ($s \leftarrow S_{1/M}$).

To illustrate the surplus normalization, consider the surplus vector obtained in the example shown in Fig. 2. The first iteration through surplus-normalization procedure gives $S_{l/M} = (-1, 1, 3, 2, 3)$; $S_{l/l+k} = (-1)$; $S_{l+k+1/M} = (1, 3, 2, 3)$. Step 4) yields $S_{l+k+1/l} = (1, 3, 2)$ and $S_{l+1/M} = (3)$ (note that $k = 0$ and $l = 4$). In step 6), a) offset = 2; b) $S_{l+k+1/M} = (-1, 1, 0, 1)$; c) $S_{l+k+1/M} = (0, 1, 0, 1)$ ("-1," the first element, is reset to 0 since it is also in $S_{l+k+1/l}$). The second iteration reassigns s_M ($M = 5$) to zero in step 5). The normalized surplus is then $s = [-1, 0, 1, 0, 0]^T$. As will be shown next (in fault shifting), based on the normalized surplus vector, the number of shifts will be $|-1|+1 = 2$ as opposed to $|-1|+1+3+2+3 = 10$ if the nonnormalized surplus vector were used.

Fault Shifting: Each unavailable cell in row i of a physical FTPA can be 1) shifted down to one of its supporters if $s_i < 0$, or 2) shifted up to one of its supporters in row j ($j < i$) if $s_j > 0$. After each shift, the corresponding entry in the surplus vector is readjusted toward zero; i.e., one is added to s_l in case 1), and one is subtracted from s_j in case 2). The shifting proceeds until the surplus vector becomes null; its effect can be described as cell migration from regions having most faulty cells to regions having less faulty cells. Fault shifting can be summarized as follows.

While $s \neq 0$ then shift unavailable cells as follows:

1) For $i = 1, 2, \cdots, M$, if $s_l < 0$ and (i, j) is unavailable, $1 \leq j \leq N + C$, then shift (i, j) to one of its supporters (if possible) in Δ_l; then increment s_l. (Shifting rules must be defined here. For instance, the selection of a good supporter could consider the closest supporter first, etc.)

2) For $i = M-1, M-2, \cdots, 1$, if $s_l > 0$ and (k, j) is unavailable, $i < k \leq M$, $1 \leq j \leq N+C$, shift (k, j) to one of its supporters (if possible) in Δ_u and in row i; then decrement s_l. (Again priority rules must be observed here.)

Fault shifting generates a status matrix for the logical FTPA. Each entry in this matrix is a status value which identifies whether the corresponding cell is fault-free, faulty, or "shifted." This value can also reveal the replacing cell location. A concrete example will be discussed in Section III.

Cell Interconnection: The cell-interconnection constructs the logical FTPA represented by the status matrix formed by "fault shifting." Fault-free cells are systematically interconnected. This implementation could be incorporated into the "fault shifting" part of the algorithm as has been done in chain-replacement schemes [4], [5]. Instead, in FUSS, fault shifting and cell interconnection are done in separate steps to facilitate the analysis of the effect of a restricted interconnection network on the algorithm. Cell interconnection will be illustrated in Section III.

D. Interconnection Requirements

This subsection shows the minimum size of a support-domain Δ required by the algorithm and the resulting maximum physical distance (thus, the maximum signal delay) between two adjacent logical cells. These requirements are for the ideal case where 100% probability of survival is guaranteed. As illustrated by simple FUSS (in Section III), survivability is not noticeably worse if interconnect is more limited and interprocessor delays are smaller.

Theorem 1: For an $M \times (N+C)$ physical FTPA with MC faulty cells ($\rho = 1$), the minimum support radius δ of each support-domain required by FUSS in order to achieve a 100% survivability rate ($\sigma = 1$) is given by

$$\delta = \left\lceil \frac{\sqrt{(2C+1)^2 + 8(M-1)C} - (2C+1)}{2} \right\rceil. \quad (2.4)$$

Proof: The worst fault distribution that causes FUSS to require the largest support-domain is a cluster (of faulty cells) that spans one corner of the physical array as shown in Fig. 4. In this figure, there are $C+1$ faulty cells in the first row. This means that one cell needs a replacement from its lower support-domain Δ_l. Let a block $(UVZXY)$ underneath the $C+1$ faulty cells contain the total possible number of additional faulty cells but such that the reconfiguration can still succeed. The maximum tolerable number of faulty cells in the FTPA is $F = S = MC$. Since there are $C+1$ faulty cells on the top row, the number of faulty cells in $(UVZXY)$ is $MC - (C+1)$. Furthermore, the $(UVZXY)$ block size (= total number of cells in the block) must be at least one cell larger than $MC - (C+1)$ so that there is at least one fault-free cell inside the block available for replacing a faulty cell in the top row. This block dictates the size of Δ_l required to support the algorithm. Its area is

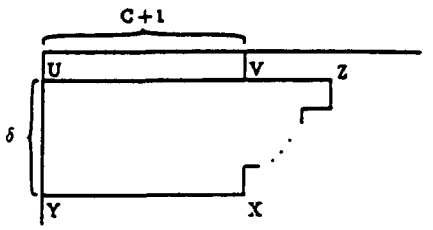
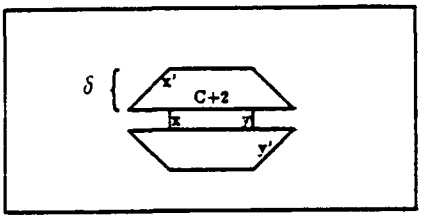
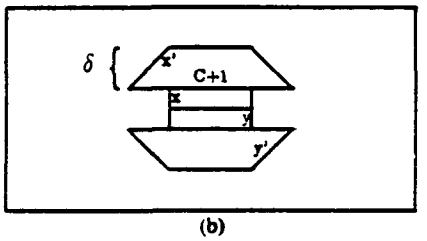


Fig. 4. Worst case block of faults.



(a)



(b)

Fig. 5. Distance between two adjacent logical cells. (a) Horizontal logical neighbors. (b) Vertical logical neighbors.

$$(UZXY) = MC - (C+1) + 1 = (M-1)C. \quad (2.5)$$

But

$$\begin{aligned}(UZXY) &= (UVXY) + (VXZ) \\ &= (C+1)\delta + (\delta - 1) + (\delta - 2) + \cdots + 1 \\ &= (C+1)\delta + \frac{\delta(\delta-1)}{2}. \end{aligned} \quad (2.6)$$

Setting (2.5) equal to (2.6), solving for δ, and taking the greatest integer value, (2.4) results. □

Theorem 2: If the support radius of the support-domain used in FUSS is δ, then the maximum possible distance between two adjacent logical cells is

$$d([i, j], [i, j+1]) = d([i, j], [i+1, j]) = 2\delta + C + 1. \quad (2.7)$$

Proof: Fig. 5(a) and Fig. 5(b) show the worst possible cases that result in largest distances between two adjacent logical cells. In these figures, the blocks shown inside each array (rectangle) contain cells that are faulty with the exception of cells marked with x' and y' which are used to replace cells x and y, respectively. In the worst

case, cell x' is at the outer left edge of the upper block shown, and cell y' is at the outer right edge of the lower block. In both figures, x' represents $[i, j]$; in Fig. 5(a), $y' = [i, j+1]$, and in Fig. 5(b), $y' = [i+1, j]$. As shown in Fig. 5(a), $d(x', y')$ is the sum of distances between x' and x $(=\delta)$, x and y $(=C+1)$, and y and y' $(=\delta)$. Therefore, $d([i, j], [i, j+1]) = 2\delta + C + 1$. The expression for $d([i, j], [i+1, j])$ is obtained in a similar way. □

III. Simple FUSS

Simple FUSS is an instance of FUSS in which each support-domain is restricted to one cell ($\delta = 1$) to minimize the maximum distance between two adjacent logical cells. In this case, shifting is either straight up or straight down along a column. From here on, the terms "simple FUSS" and "FUSS" are used interchangeably; simple FUSS with $C = 1$ is referred to as "FUSS-1," that with $C = 2$ "FUSS-2," and so on. The general FUSS scheme discussed in the last section is referred to as "general FUSS."

A. Simple FUSS Algorithm

The algorithm consists of four basic procedures, each of which describes a corresponding part in the general FUSS algorithm described in Section II. Procedures corresponding to array preprocessing and surplus normalization are as described in Section II. In fault shifting, cell (i, j) can only replace either cell $(i-1, j)$ or cell $(i+1, j)$. Fault shifting constructs status matrix B as follows:

1) Initialize B: $b_{i,j} = \begin{cases} 0 & \text{if } (i, j) \text{ is fault-free} \\ 1 & \text{if } (i, j) \text{ is faulty} \end{cases}$.

2) Assign status to "shifted-cells": $b_{i,j} = 2$ if cell $(i+1, j)$ is shifted to cell (i, j), and $b_{i,j} = 3$ if cell $(i-1, j)$ is shifted to cell (i, j).

Matrix B is formed one column at a time. The order in which the columns are considered may affect the shift capability. It is therefore conceivable that these columns can be scanned in an order that maximizes FUSS probability of survival but this remains an open question at this time. For simple FUSS, the order is from columns 1 to $N+C$.

Cell interconnection is applied to matrix B. Since the status of each cell is known, it is easy to automatically derive how cells are interconnected. FUSS-C algorithm is shown in detail in [7]. Each of the four procedures in the general FUSS algorithm takes at most $O(M \times (N+C))$ operations. The time complexity of FUSS is then $O(MN)$, i.e., proportional to the number of processors in the array.

The following example illustrates how FUSS-1 reconfigures a $7 \times (5+1)$ array. Fig. 6(a) represents the initial status matrix of the FTPA augmented by the normalized surplus vector; row and column indexes are added for convenience. Fault shifting scans s downward for negative surplus values. Since $s_4 = -1$, one cell is shifted down; the only choice in this case is (4, 4). As a result $b_{5,4}$ becomes 3, to denote that (5, 4) replaces a cell from above, namely (4, 4). At the next row, $s_5 = -2$, which means that two unavailable cells must be shifted down; cells (5, 2) and (5, 3) are chosen. Similarly, $s_6 = -1$ and (7, 2) replaces (6, 2). During the upward scan, the first positive surplus encountered is $s_3 = 1$. Thus, one cell from the row below must be shifted upward to row 3. Cell (4, 2) is the only choice; as a result, $b_{3,2} = 2$ to denote that cell (3, 2) replaces a cell from below. Similar actions take place for s_2 and s_1. Fig. 6(b) shows the final status matrix after a successful cell shifting (s is null). The final reconfigured FTPA is shown in Fig. 6(c). This example depicts the worst case of fault distribution in a physical array that results in longest interconnect links in a logical array.

B. Cell Interconnect Structure

From Theorem 2, the maximum distance between two adjacent logical cells is $d([i, j], [i, j+1]) = d([i, j], [i+1, j]) = C + 3$. Fig. 6(c) shows that the distance between [4, 2] and [4, 3] is 4 ($\phi([4, 2]) = (3, 2)$ and $\phi([4, 3]) = (5, 4)$). The distance between [3, 3] and [4, 3] is also 4. In general, the interconnect requirements for FUSS can be found by studying the connection windows required

	1	2	3	4	5	6	s
1	0	0	0	0	0	0	1
2	0	0	0	0	0	0	2
3	0	0	1	1	0	0	1
4	0	1	1	1	0	0	-1
5	0	1	1	0	0	0	-2
6	0	0	0	0	0	0	-1
7	0	0	0	0	0	0	0

(a)

	1	2	3	4	5	6	s
	0	2	0	0	0	0	0
	0	2	2	0	0	0	0
	0	2	1	1	0	0	0
	0	1	1	1	0	0	0
	0	1	1	3	0	0	0
	0	3	3	0	0	0	0
	0	3	0	0	0	0	0

(b)

1,1	2,2	1,2	1,3	1,4	1,5
2,1	3,2	3,3	2,3	2,4	2,5
3,1	4,2	0,0	0,0	3,4	3,5
4,1	0,0	0,0	0,0	4,4	4,5
5,1	0,0	0,0	4,3	5,4	5,5
6,1	5,2	5,3	6,3	6,4	6,5
7,1	6,2	7,2	7,3	7,4	7,5

(c)

Fig. 6. An example of FUSS-1. (a) Initial status matrix and surplus vector $B|s$. (b) Final status matrix B (s = 0). (c) Final reconfigured FTPA showing logical coordinates.

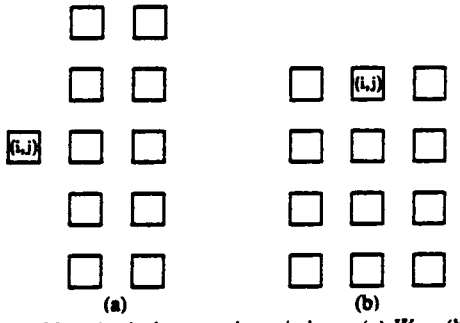


Fig. 7. FUSS-1 physical connection windows. (a) W_H. (b) W_V.

by the algorithm. The size of each window (number of cells in the window) is as stated in the following theorem.

Theorem 3: In FUSS, the size of horizontal connection window $|W_H|$ and the size of vertical connection window $|W_V|$ of a cell are given by, respectively,

$$|W_H| = 5(C+1) \tag{3.1}$$

$$|W_V| = 4(2C+1) - 1. \tag{3.2}$$

Proof: From cell shifting, $\phi_l([i, j]) = k$, $i-1 \le k \le i+1$ ($[i, j]$ is on physical row k), and $\phi_l([i, j+1]) = l$, $i-1 \le l \le i+1$. This means that the physical location of $[i, j+1]$ is at most two physical rows above or two rows below the physical location of $[i, j]$. Therefore, relative to $\phi([i, j])$, $[i, j+1]$ is within a band consisting of five physical rows as shown in Fig. 7(a) (for $C = 1$). Since there are C spare columns, the link from $[i, j]$ to $[i, j+1]$ can skip over at most a number of C cells. Hence, $|W_H| = 5(C+1)$.

Without loss of generality, let $\phi([i, j]) = (i, j)$. Consider the physical location of the vertical neighbor $[i+1, j]$. Since physical cells may be shifted up from the immediate row below, $[i+1, j]$ can be on physical row i; i.e., $\phi_l([i, j]) = \phi_l([i+1, j]) = i$. Also since cells may be shifted down, the physical location of $[i+1, j]$ may be three physical rows below that of $[i, j]$ ($\phi_l([i, j]) = \phi_l([i-1, j]) - 3$). Furthermore, $[i+1, j]$ can be C cells to the left (right) of $[i, j]$. The vertical window is readily seen to have the size of $|W_V| = 4(2C+1) - 1$. □

For FUSS-1, $|W_H|$ and $|W_V|$ are 10 and 11, respectively, as shown in Fig. 7. This means that interconnect links must be provided for each cell so that it can communicate with any one of 10 cells horizontally and any one of 11 cells vertically. These are not necessarily dedicated links; they can be shared buses, virtual channels, etc. In this paper a switch-bus similar to that used in [5]–[7] is considered. In general, the number of buses and switches required by a logical row is constant and that required by a logical column is a linear function of the number of spare columns. FUSS-C requires a $(\alpha+1) \times 3$

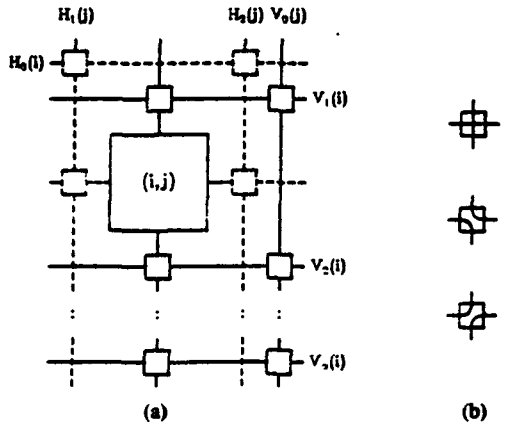


Fig. 8. Switch-bus structure. (a) FUSS-C. (b) Switch settings.

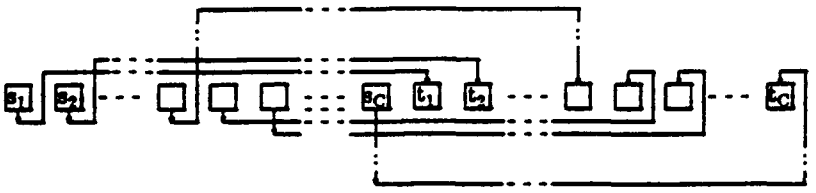


Fig. 9. Source-sink on same row; source at left.

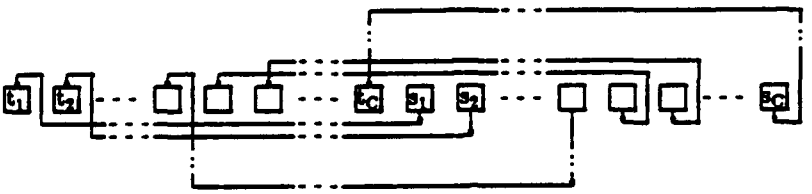


Fig. 10. Source-sink on same row; source at right.

switch bus as shown in Fig. 8(a), where $\alpha = \lceil 1.5C \rceil$. (Thus, FUSS-1 requires a 3×3 switch bus.) The following corollaries give the size of the interconnection.

Corollary 1: In addition to cell-to-cell links, the horizontal path of FUSS consists of one horizontal bus per row, two vertical buses per column, and four switches per cell of the physical FTPA.

Corollary 2: Let $\alpha = \lceil 1.5C \rceil$. In addition to cell-to-cell links, the vertical path of FUSS consists of one vertical bus per column, α horizontal buses per row, and 2α switches per cell of the physical FTPA.

Fig. 8 shows a switch-bus structure required by FUSS. With this structure, there are many possible paths between any pair of adjacent logical cells. Therefore, a shortest path must be devised so that the interconnect delay is minimized. Due to space limitations, the following bypass interconnect rules are summarized next (details are given in [7]). Bus labels H_i, $i = 0, 1, 2$, and V_i, $i = 0, 1, \cdots, \alpha$ are as shown in Fig. 8. Let $\phi([m, n]) = (i, j)$, $\phi([m, n+1]) = (k, l)$, and $\phi([m+1, n]) = (p, q)$; $i-2 \le k \le i+2$, $j+1 \le l \le j+C+1$, $i \le p \le i+3$, and $j-C \le q \le j+C$. Cell $[m, n]$ is called source (denoted by s) and cells $[k, l]$ and $[p, q]$ are horizontal sink and vertical sink, respectively (denoted by t).

Connecting (i, j) to (k, l):

- use $H_0(i)$ to go rightward;
- use $H_1(j+1)$ or/and $H_1(l)$ to go downward;
- use $H_2(j)$ or/and $H_2(l-1)$ to go upward.

Connecting (i, j) to (p, q):

Let $\beta = \lceil 0.5\alpha \rceil$, where $\alpha = \lceil 1.5C \rceil$.

1) If $i < p$ (Fig. 11), then
   - use the first available $V_x(i)$, $x = 2, 3, \cdots, \beta+1$ to go right or left toward (p, q);
   - use $V_0(q)$ to go downward;
   - use $V_1(p)$ to reach (p, q).
2) If $i = p$, then
   a) If $j < q$ (source on the left; Fig. 9):
      i) use the first available bus from the ordered set $\{V_1(i), V_\alpha(i-1), V_{\alpha-1}(i-1), \cdots, V_{\beta+2}(i-1)\}$ to go right;
      ii) if all buses in case i) are exhausted, use the first available bus from the ordered set $\{V_2(i), V_3(i), \cdots, V_{\beta+1}(i)\}$ to go right;
      iii) use $V_0(j)$ for case i), $V_0(q)$ for case ii)) to go upward.
   b) If $q < j$ (source to the right of sink; Fig. 10):
      i) use the first available bus from the ordered set $\{V_2(i), V_3(i), \cdots, V_{\beta+1}(i)\}$ to go left;
      ii) if all buses in case i) are exhausted, use the first available bus from the ordered set $\{V_1(i), V_\alpha(i-1), V_{\alpha-1}(i-1), \cdots, V_{\beta+2}(i-1)\}$ to go left;
      iii) use $V_0(q)$ for case i), $V_0(j)$ for case ii)) to go upward.

Illustrations of interconnection between $(i, j) = s$ and $(p, q) = t$ are shown in Figs. 9–11 (common subscript is used for a source-sink pair). Bus V_0 is exclusively assigned to its corresponding cell. It directs the signal downward for case 1) and upward for case 2). This exclusive assignment of V_0 may result in extra wire length (two units). For instance, in Fig. 9, V_0 on the left of t_C (and V_0 for some other sinks) may be used. However, this can make the rules more complex.

C. FTPA Structure and Area Overhead

The FTPA control mechanisms required to support a reconfiguration scheme can be implemented in different ways. Since a scheme can be designed to be either "centralized" (executed by a central computer) or "distributed," the complexity of the control hardware varies accordingly. The interconnect redundancy depends on whether

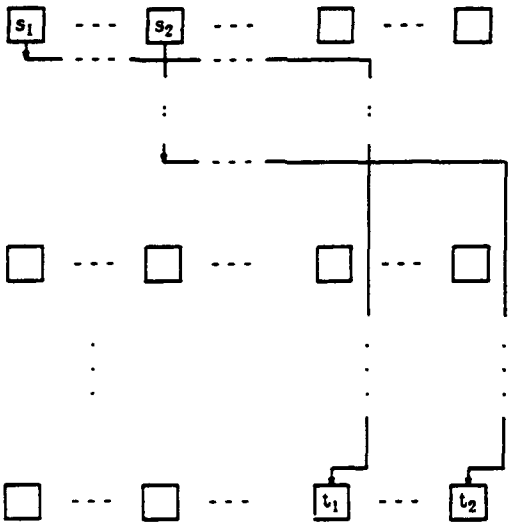


Fig. 11. Source-sink on different row.

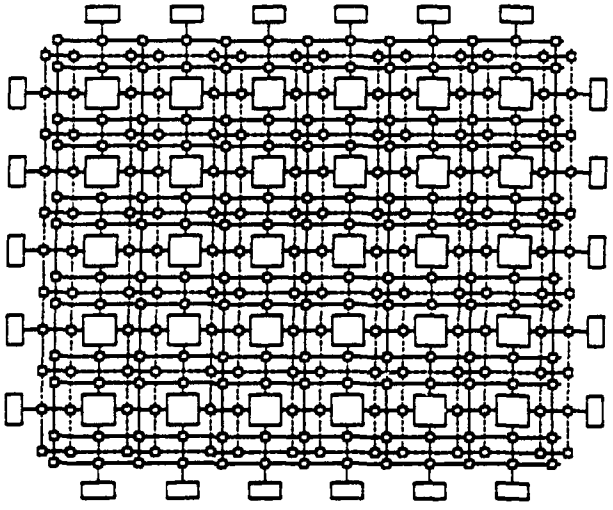


Fig. 12. FUSS-1 FTPA.

multiplexers, virtual channels, switched buses or other approaches are used. FUSS-C uses the switch-bus structure shown in Fig. 8 and a typical FTPA is shown in Fig. 12, for FUSS-1. In Fig. 12, rectangles represent I/O ports, large boxes represent cells, small boxes represent switches, dashed lines are buses for horizontal paths, and solid lines buses for vertical path. Other reconfiguration circuitry (fault and surplus counters) is not shown in Fig. 14 but can be found in [7] where area complexity is also computed.

D. FUSS Performance

FUSS has been evaluated with respect to its probability of survival and its reliability. The former was achieved by extensive simulation and the latter was estimated by the MGRE (model generator and reliability evaluator) software tool [9] (details shown in [7]).

Arrays of various sizes (5×5 to 40×40) and with varying number of spare columns ($C = 1$–$C = 5$) have been simulated. Faults were randomly injected into the array cells. In each case, probability of survivals are obtained for 100 000 arrays having demand-spare $0.5 \leq \rho \leq 1.0$. When $\rho < 5$, the survivability rate is 100%; it is more than 95% when $\rho = 1$. For the purpose of comparison, probabilities of survival of a number of schemes are shown, along with the complexity of each algorithm, in Table I for $M = N = 20$.

FUSS survivability rate can be further improved by slightly modifying the cell-shifting procedure. In the new cell-shifting approach, cell (i, j) is shifted to cell $(i \pm 1, j)$, not only when cell $(i \pm 1, j)$ is available, but also (if possible) when cell $(i \pm 2, j)$ is available.

TABLE I
PROBABILITY OF SURVIVAL OF SELECTED RECONFIGURATION SCHEMES (PARTIALLY FROM [3], [4])

Probability of Survival σ of FUSS with Improved Cell-Shifting Number of Faults = Number of Spares ($\rho = 1$)					
C	For Array Size ($M \times N$)				
	5×5	10×10	20×20	30×30	40×40
1	99.2567	98.7047	98.7383	98.9145	99.0524
2	98.7698	98.6465	99.1754	99.4904	99.6391
3	98.4911	98.6554	99.4346	99.7239	99.8534
4	98.3180	98.6508	99.5387	99.8301	99.9214
5	98.1736	98.6242	99.6006	99.8670	99.9509

TABLE II
ARRAY SURVIVABILITY AT MAXIMUM NUMBER OF FAULTS

Reconfiguration Scheme	Algorithm Complexity	Probability of Survival		
		$\rho = .50$	$\rho = .75$	$\rho = .90$
Direct Reconfiguration [10]	$O(N)$	0.43	0.35	0.27
Fixed-Fault-Stealing	$O(N)$	0.47	0.38	0.30
Complex-Fault-Stealing	$O(N)$	0.90	0.84	0.77
Modified Index-Mapping	$O(N^2)$	0.67	0.62	0.54
MORA	$O(N \log N)$	0.90	0.84	0.78
Spanning Tree Based	$O(N \log \log N)$	0.93	0.89	0.78
Orthogonal Mapping	$O(N^2)$	0.93	0.88	0.82
Simple FUSS	$O(N^2)$	1.00	1.00	0.99

This provides a one cell "look-ahead" capability in anticipating and facilitating further shifting. Simulation results show much improved probability of survival as summarized in Table II. Here, only the worst case ($\rho = 1$, number of faults equal to number of spares) is shown. Each case involves 1 000 000 simulated arrays.

IV. CONCLUDING REMARKS

A new approach to hardware reconfiguration for fault-tolerant processor arrays has been described. The technique, called FUSS (full use of suitable spares), is based on global information about faulty cells in the array. That is, the number of faulty cells and their distribution in the array are compiled to form a surplus vector which is used to guide cell replacement. The general FUSS can achieve 100% probability of survival when unlimited interconnect links are provided. These interconnection requirements have been derived and shown to be proportional to the size of the array.

An instance of FUSS, called simple FUSS, has been presented in detail. Simulations show that simple FUSS achieves up to 99% probability of survival when the number of faulty cells equals the number of spare cells in a given array. Simple FUSS executes in $O(MN)$ time, for an $M \times (N + C)$ array.

Recently, Youn and Singh [11] have independently proposed an approach, called row modular scheme, similar to simple FUSS (the FUSS approach was first reported in [12]). The approach is an instance of general FUSS and is implemented for yield enhancement in WSI/VLSI processor arrays.

The FUSS approach can be readily applied to other mesh-like structures, such as programmable logic arrays, memory arrays, etc. While FUSS has been designed for two-dimensional arrays, it may also be extended to other architectures such as modified meshes (toroidal meshes, meshes with broadcast buses, etc.), trees, hypercubes, multistage networks, and others. The use of FUSS in these and other useful topologies needs further study and evaluation [7].

REFERENCES

[1] M. Chean and J. A. B. Fortes, "A taxonomy for reconfiguration techniques for fault-tolerant processor arrays," *IEEE Computer Mag.*, Jan. 1989.

[2] R. Negrini and R. Stefanelli, "Comparative evaluation of space- and time-redundancy approaches for WSI processing arrays," in *Wafer Scale Integration*, G. Saucier and J. Trilhe, Eds., New York: Elsevier Science, 1986, pp. 207–222.

[3] L. Jervis, F. Lombardi, and D. Sciuto, "Orthogonal mapping: A reconfiguration strategy for fault tolerant VLSI/WSI 2-D arrays," in *Proc. Int. Workshop. Defect Tolerance VLSI Syst.*, Oct. 1988.

[4] F. Lombardi, D. Sciuto, and R. Stefanelli, "A technique for reconfiguring two dimensional VLSI arrays," in *Proc. Real-Time Syst. Symp.*, 1987, pp. 44–53.
[5] F. Lombardi, R. Negrini, and R. Stefanelli, "Reconfiguration of VLSI arrays: A covering approach," in *Proc. 17th Int. Symp. Fault-Tolerant Comput. Syst.*, 1987, pp. 251–256.
[6] M. G. Sami and R. Stefanelli, "Fault-tolerance and functional reconfiguration in VLSI arrays," in *Proc. Int. Symp. Circuits Syst.*, 1986, pp. 643–648.
[7] M. Chean, "Hardware reconfiguration for fault-tolerant processor arrays," Ph.D. dissertation, School of Elec. Eng., Purdue Univ., 1989.
[8] L. Snyder, "Introduction to the configurable highly parallel computers," *IEEE Comput. Mag.*, vol. 15, pp. 47–55, Jan 1982.
[9] N. Lopez-Benitez and J. A. B. Fortes, "Detailed modeling of fault-tolerant processor arrays," in *Proc. 19th Int. Symp. Fault-Tolerant Comput. Syst.*, June 1989, pp. 545–552.
[10] M. G. Sami and R. Stefanelli, "Reconfigurable architectures for VLSI processing arrays," *Proc. IEEE*, vol. 74, pp. 712–722, May 1986.
[11] H. Y. Youn and A. D. Singh, "Efficient reconfiguration of WSI arrays with bounded channel width," in *Proc. Int. Workshop. Hardware Fault Tolerance Multiprocessors*, June 1989, pp. 24–26.
[12] M. Chean and J. A. B. Fortes, "FUSS: A reconfiguration scheme for fault-tolerant processor arrays," in *Proc. Int. Workshop. Hardware Fault Tolerance Multiprocessors*, June 1989, pp. 30–32.

On Reliability Modeling of Closed Fault-Tolerant Computer Systems

MEERA BALAKRISHNAN AND C. S. RAGHAVENDRA

Abstract—**Markov modeling techniques have proved useful for reliability prediction of complex fault-tolerant computers. In [7], Ng and Avizienis proposed a general model for evaluating fault-tolerant computer systems—a continuous-time Markov model referred to as the ARIES model and suggested the Lagrange-Sylvester interpolation formula as a unified solution technique to this model. This is a valid solution approach if the eigenvalues of the state transition rate matrix associated with the Markov chain are distinct or, if eigenvalues repeat, the rate matrix is diagonalizable. In [2], Geist and Trivedi have presented an example of a closed (i.e., nonrepairable) fault-tolerant system modeled by ARIES which has repeated eigenvalues; in this paper, we observe that, in fact, a large number of closed fault-tolerant systems modeled by ARIES have repeated eigenvalues. The Lagrange-Sylvester formula is an acceptable solution approach only if it can be guaranteed that the rate matrix is diagonalizable.**

In this paper, we prove that the rate matrix representing the system is diagonalizable for every *closed* fault-tolerant system modeled by ARIES. Consequently, the Lagrange-Sylvester interpolation formula is applicable to all closed fault-tolerant systems which ARIES models. Also, since our proof guarantees that the rate matrix is diagonalizable, general methods for solving arbitrary Markov chains can be tailored to solve the ARIES model for closed systems efficiently. For example, the ACE algorithm for the transient solution of acyclic Markov chains [4] can be specialized to solve the ARIES model for closed systems efficiently.

Index Terms—**Closed fault-tolerant systems, Laplace transform, Markov modeling, reliability prediction.**

Manuscript received July 3, 1989; revised November 6, 1989. M. Balakrishnan was supported by a Zonta International Amerlia Earhart Fellowship Award. C. S. Raghavendra was supported by the NSF Grant MIP8452003, a grant from AT&T, and a grant from TRW.

The authors are with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 8933878.

I. Introduction

A major application of closed (nonrepairable) fault-tolerant computer systems is in the aerospace industry. Indeed, it was the space program in the early years that motivated research in the area of fault-tolerant computers. In [2] the term *ultrahigh-reliability* has been used to indicate the stringent reliability requirements demanded by these nonrepairable computer systems. With such demands on the reliability measure, mathematical modeling is often the only means of assessing and comparing the reliability of two or more ultra-reliable systems.

Several mathematical models for reliability prediction have been developed and are available as software packages ([1], [3], [9]). These packages accept a high-level description of a fault-tolerant system and internally generate an equivalent mathematical model of the system. Various dependability measures are then derived by solving the mathematical model. For a review and critical evaluation of several of these models, we refer the reader to [2] and [5]. The ARIES model proposed by Ng and Avizienis [7] is a unified reliability model developed for evaluating fault-tolerant computer systems. Based on the Markov modeling technique, it unifies most reliability models proposed in the 1970's. Further, the solution for the reliability function $R(t)$ is an analytical expression which is in the form of a weighted sum of pure negative exponentials. Hence, the solution is completely specified by a set of parameter pairs (parameters of the exponentials and the corresponding multipliers) which depend on the fault-tolerant system under evaluation. However, the solution is of this form only if the state transition rate matrix of the Markov chain is diagonalizable.

Let $\mathcal{B}$ denote the transition rate matrix. The solution approach in ARIES is to compute the matrix function $e^{\mathcal{B}t}$ using the Lagrange-Sylvester interpolation formula. The state probability vector is then given by $P(t) = e^{\mathcal{B}t}P(0)$ and the reliability function by $R(t) = \sum_{i \in \text{states}} P_i(t)$. From standard matrix theory, if the eigenvalues of $\mathcal{B}$ are distinct, then, $\mathcal{B}$ is diagonalizable so that the minimal polynomial of $\mathcal{B}$ has distinct roots. For this case, the Lagrange-Sylvester formula[1] is a valid solution technique. If the eigenvalues are not distinct, however, $\mathcal{B}$ may or may not be diagonalizable. For this case of repeated eigenvalues, ARIES considers only the distinct eigenvalues (ignoring duplicates) and solves as in the previous case. Since this is a valid approach only when $\mathcal{B}$ is diagonalizable, it is not clear that the ARIES results are correct for all systems with repeated eigenvalues. In [2], Geist and Trivedi have given an example of an ARIES closed fault-tolerant system in which the eigenvalues of the transition rate matrix are not distinct. Further, they have observed that the set of eigenvectors of the transition rate matrix are independent (i.e., $\mathcal{B}$ is diagonalizable) and that classical solution methods yield correct results. In this, paper we prove that for ARIES closed systems, $\mathcal{B}$, the transition rate matrix, is always diagonalizable so that the Lagrange-Sylvester formula is applicable despite repeated eigenvalues. Further, since our proof guarantees that the rate matrix is always diagonalizable, general methods for solving arbitrary Markov chains can be tailored to solve the ARIES model efficiently.

II. ARIES Model and Solution Technique for Closed Fault-Tolerant Systems

We begin with a brief description of the ARIES model for closed fault-tolerant systems (for a detailed description, see [6] and [7]). The input to ARIES consists of a set of parameters which completely specifies the fault-tolerant system. The definition of these parameters [6] is reproduced here for easy reference.

N = Initial number of modules in the active configuration
S = Number of spare modules

[1] This formula, described by (2.2) in this paper, is applicable only if $\mathcal{B}$ is diagonalizable.

0018-9340/90/0400-0571$01.00

REFERENCE NO. 8

Lopez-Benitez, N. and Fortes, J. A. B., "Detailed Modeling and Reliability Analysis of Fault-Tolerant Processor Arrays," *IEEE Transactions on Computers*, Vol. 41, No. 9, September 1992, pp. 1193-1200.

Note - This paper presents an efficient approach to the problem of modeling and evaluating the reliability of fault-tolerant processor arrays. It also reports results obtained by using a software package that implements the proposed approach.

[4] T. Kløve, "Error correcting codes for the asymmetric channel," Rep. 18-09-07-81, Dep. Mathematics, Univ. Bergen, July 1981.
[5] D. J. Lin and B. Bose, "Theory and design of t-error correcting and d ($d > t$) unidirectional error detecting (t-EC d-UED) codes," *IEEE Trans. Comput.*, vol. C-37, pp. 433–439, Apr. 1988.
[6] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam, The Netherlands: North-Holland, 1977.
[7] R. J. McEliece, *The Theory of Information and Coding*. Reading, MA: Addison-Wesley, 1977.
[8] D. Nikolos, "Theory and design of t-error correcting/d-error detecting ($d > t$) and all unidirectional error detecting codes," *IEEE Trans. Comput.*, vol. C-40, pp. 132–142, Feb. 1991.
[9] D. Nikolos, N. Gaitanis, and G. Philokyprou, "Systematic t-error correcting/all unidirectional error detecting codes," *IEEE Trans. Comput.*, vol. C-35, pp. 394–402, May 1986.
[10] D. K. Pradhan and S. M. Reddy, "Fault-tolerant failsafe logic networks," in *Proc. IEEE COMPCON*, San Francisco, CA, Mar. 1977, p. 363.

Detailed Modeling and Reliability Analysis of Fault-Tolerant Processor Arrays

N. Lopez-Benitez and J. A. B. Fortes

Abstract—**A method for the generation of detailed models of fault-tolerant processor arrays, based on Stochastic Petri Nets (SPN) is presented in this paper. A compact SPN model of the array associates with each transition a set of attributes that includes a discrete probability distribution. Depending on the type of component and the reconfiguration scheme, these probabilities are determined using simulation or closed-form expressions and correspond to the survival of the array given that a number of components required by the reconfiguration process are faulty.**

Index Terms—**Fault-tolerance, Markov models, processor arrays, reliability, stochastic Petri nets.**

I. Introduction

As is the case with many systems, Markov models can be used to evaluate the reliability of processor arrays. However, reliability estimations are mostly based on the failures of processing elements only [1]. Components other than processing elements become very important in the analysis of fault-tolerant processor arrays because of their susceptibility to faults and the added hardware complexity of the overall array. This fact has played an important role in the derivation of the mathematical framework developed by Koren *et al.* [2] to evaluate yield improvement and performance-related measures of different array architectures. A detailed modeling of fault-tolerant processor arrays, which explicitly takes into consideration the failure statistics of each component as well as their possible interdependencies, entails not only an explosive growth in the model state space but also a difficult model construction process. This paper proposes a systematic method to construct Markov models for evaluating the reliability of processor arrays. The method is based on the premise that the fault behavior of a processor array can be modeled by a Stochastic Petri Net (SPN). However, in order to obtain a more compact and efficient representation, a set of attributes is associated with each transition in the Petri net model. This set of attributes allows the construction of the corresponding Markov model as the reachability graph is being generated. Included in these attributes is a discrete probability distribution such that the effect of faulty spares in the reconfiguration algorithm is captured each time a configuration change occurs. This distribution includes the probabilities of survival given that a number of components required by the reconfiguration process are faulty. Depending on the type of component and the reconfiguration scheme, these distributions are determined using simulation or closed-form expressions. The application of this method is illustrated by the detailed analysis of the SRE reconfiguration scheme [3]; also, analytical expressions to estimate probabilities of survival are derived for this scheme. Other applications that include schemes such as ARCE (Alternate Row-Column Elimination) [3] and DR (Direct Reconfiguration) [4], are reported in [5].

Once the Petri net model and the corresponding reachability graph have been obtained, all the information required to build the transition matrix of the corresponding Markov chain is available. Reliability evaluation tools such as ARIES [6] and SHARPE [7] can be used to evaluate the models developed here.

The second section of this paper discusses some basic notation and concepts which include array configurations and Petri nets; also, the large state space inherent in the models generated is illustrated. The third section discusses MSPN models as an extension of SPN's. Throughout these two sections the SRE reconfiguration scheme is used as an application. In the fourth section a procedure used in generating the reachability graph is described. Finally, results on reliability analysis are reported in Section V.

Manuscript received September 11, 1990; revised October 21, 1991. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contracts 00014-85-k-0588 and 00014-88-k-0723.

N. Lopez-Benitez is with the Department of Electrical Engineering, Louisiana Tech University, Ruston, LA 71272.
J. A. B. Fortes is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 9200308.

II. Preliminaries

A. Array Configurations

To analyze a fault-tolerant array architecture with k types of components, the configuration of an array is represented as a k-tuple:

$$C_i = (\eta_{1i}, \eta_{2i}, \cdots, \eta_{ki}) \qquad i = 0, 1, \cdots, |C|$$

where η_{li} denotes the number of elements of component type l and C is the set of all possible configurations of the array. Examples of component types include processing elements, links, switches, spare links, and spare processing elements. The occurrence of faults and the application of the reconfiguration algorithm define a sequence of configurations that begins with C_0 as the *initial configuration*; any other configuration can correspond to the failure state or an operational state of the array. The latter will be referred to as an *operational configuration*.

Upon detection of a faulty component, the reconfiguration algorithm may not send the array to an operational configuration if any of the following happens:

1) The reconfiguration circuitry failed. This possibility can be considered through a *coverage factor* (denoted by c) defined as the probability of successful reconfiguration given that a fault has occurred [8]. This is a measure of the probability of successful operation of all circuitry related to fault detection, isolation, and reconfiguration. The coverage factor is assumed constant and it will be associated with failures of active components only.

2) Redundancy is exhausted. This information can be inferred from C_i.
3) The presence of faults in nonactive components (redundancy) hinders a successful reconfiguration. Redundant components are present in C_i as spare processing elements, spare switches, spare links, spare buses, etc. Some of these components become active in the new configuration.

In a given configuration with a number of faulty components, a successful reconfiguration, will depend not only on the type of faults but also on their distribution in the array. Thus, the probability of a correct reconfiguration in the presence of faults is referred to as the *probability of survival* [4]. Because the reconfiguration algorithm may choose one of several new configurations (including a nonoperational one), a probability is assigned to each possible new configuration. The probability of survival corresponds to the sum of probabilities assigned to new operational configurations.

B. The Largeness Problem

Consider for example an $n \times n$ array that supports the Successive-Row-Elimination (SRE) reconfiguration scheme with a layout as in Fig. 1 (where $n = 4$). A row of Switches (S) and Spare Bypass Links bypass all the Processing Elements (PE's) in any row containing at least one faulty PE or at least one faulty horizontal link (HL) or at least one faulty input/output link (IOL); spare bypass links (SBL's) then become active Bypass Links (BL); the array performance degrades gracefully as rows are eliminated. The minimum working configuration consists of a single functional row of PE's. However, all active bypass links and all switches must be fault-free. The array fails if operational rows are exhausted or a single switch or an active bypass link have failed. Assuming only PE's fail, a Markov chain will contain n operational states [3]. For each operational state i in the Markov chain there will be one transition to an operational state j with a rate denoted as λ_{ij} and one transition to the failure with a rate λ_{if}. If the failure rate of a PE is α_1, then these transition rates can be evaluated as $\lambda_{ij} = \eta_{i1}\alpha_1 c$ and $\lambda_{if} = \eta_{i1}\alpha_1(1-c)$ where η_{i1} corresponds to the number of PE's active when the process is in state i and c is a constant coverage factor.

Assume now that not only PE's fail but also switches are susceptible to failure with a rate α_3. Note that the failure of a single switch will cause the array to fail; therefore, the transition rate to the failure state is modified accordingly; i.e., $\lambda_{if} = \eta_{i1}\alpha_1(1-c) + \eta_{i3}\alpha_3$ where η_{i3} corresponds to the number of switches when the process is in state i. In addition to failures of PE's and switches, assume failures of links including spare and active bypass links can occur. Failures of I/O links and horizontal links can be treated as PE failures in the sense that the same state will be generated but with a different transition rate. Of particular interest is the failure of spare bypass links, because failures of this type will generate a large state space. For $n = 4$ the number of states increases to 44 as shown in Table I. Although it is possible to construct the model manually, the process has now become complex and time consuming. In addition, nondistinct states must be merged and the *survivability* and *coverage* factors must be aggregated to estimate transition rates.

The maximum number of states $S_{SRE}(n)$ generated in terms of the size n of the array is tabulated in Table I. An expression of $S_{SRE}(n)$ is obtained by observing that n states are generated with nonfaulty spare bypass links; for each operational state i, $n(n-i)$ additional states can be generated with at least one faulty spare bypass link. Hence, $S_{SRE}(n) = \sum_{i=1}^{n-1}[n(n-i)+1]$. Simplifying this expression, the maximum number of states is given by

$$S_{SRE}(n) = \frac{1}{2}(n^3 + n^2 + 2n).$$

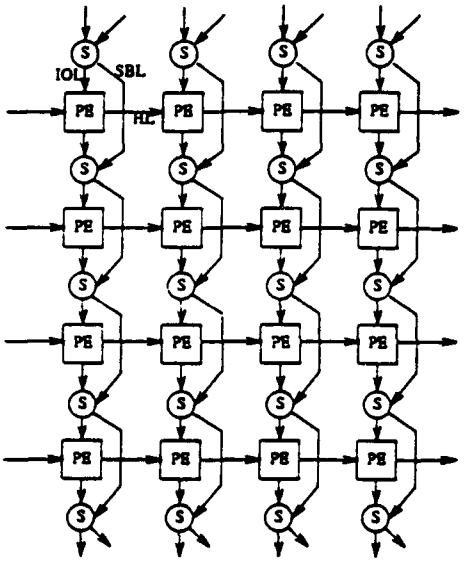


Fig. 1. Schematic layout of the SRE structure.

Similar expressions have been derived for the ARCE and DR schemes [5] whose values are shown in Table I for several instances of n.

The complexity involved in constructing Markov chains with detailed modeling obviates the need for a higher level representation such as the Modified Stochastic Petri Nets proposed in this paper. This type of modeling is useful in representing the fault behavior of the array in the presence of different types of faults and in generating the Markov chain for any array size and in the format required by the chosen evaluation package.

C. Petri Nets

Petri nets can be used to analyze complex systems with interdependent components. A formal definition of a Petri net is presented in [9] as a 4-tuple $PN = (P, T, A, M_0)$ where $P = \{p_1, p_2, \cdots, p_k\}$ is the set of places, $T = \{t_1, t_2, \cdots, t_r\}$ is the set of transitions, $A \subset \{P \times T\} \cup \{T \times P\}$ is the set of input and output arcs and M_0 is the initial marking. Each place contains a number of *tokens*. A *marking* in a PN is represented by a k-component vector $M = [m_1, m_2, \cdots, m_k]$ where m_i is the number of tokens in place p_i. The set of input places I and the set of output places O are mappings defined with respect to a transition t_i as $I(t_i) = \{p | (p, t_i) \in A\}$ and $O(t_i) = \{p | (t_i, p) \in A\}$, respectively. The number of tokens associated with an arc is referred to as the *multiplicity* of the arc. A transition is *enabled* when a PN is in marking M such that all input places of t_i contain a number of tokens greater than or equal to the multiplicities associated with the corresponding arcs. A transition *fires* if it is enabled. When a transition fires, tokens are removed from its input places and added to its output places according to the multiplicities of the corresponding arcs. This determines a *new marking*. The *reachability set* of a PN is the set of all markings reachable from M_0 through a sequence of firings. The *reachability graph* of a PN is a directed graph whose vertices are the elements of the reachability set and whose arcs correspond to transition firing in the PN.

III. Modified Stochastic Petri Nets

A. SPN Representation

Stochastic Petri Nets extend the Petri Net concept by associating random firing times with each transition. This extension allows a one-to-one mapping between the markings generated by a Stochastic Petri Net and the operational states of an homogeneous Markov chain

TABLE I
MAXIMUM STATE SPACE SIZE FOR THREE RECONFIGURATION SCHEMES

n	2	3	4	5	6	7	8	9	10
$S_{SRE}(n)$	8	21	44	80	132	203	296	414	560
$S_{ARCE}(n)$	17	51	107	199	333	517	759	1067	1449
$S_{DR}(n)$	75	188	385	690	1127	1720	2493	3470	4675

(assuming exponentially distributed firing times) [10]. Thus, SPN's are defined by a 5-tuple (P, T, A, M, α) where $\alpha = \{\alpha_1, \alpha_2, \cdots, \alpha_s\}$ specifies the firing rates of transitions $t_1, t_2, \cdots, t_s$. To model fault-tolerant processor arrays, an operational configuration corresponds to an operational state in the Markov chain; thus, to derive all possible operational configurations of the array, a marking in the SPN must correspond to an operational configuration of the array. Each place p_i identifies components of type i and, at a given marking M_q, the number of tokens m_{iq} corresponds to n_{iq} which is the number of components of type i. Two or more distinct component types may identify the same physical component; for example, a physical spare is a component of the type "active spare" when it is used to replace or bypass a faulty part and it is a component of the type "nonactive spare" otherwise. Consider the $n \times n$ SRE array in Fig. 1; a marking M_q is described as $M_q = (\#PE, \#IOL, \#S, \#BL, \#HL, \#SBL) = (m_{1q}, m_{2q}, \cdots, m_{6q})$ where the symbol "#" is used to denote "number of."

A possible SPN representation is given in Fig. 2. The firing of t_1 represents the occurrence of a fault in a PE, a fault in an IOL is represented by the firing of t_2 and so on. In general, the firing of t_i represents a fault occurrence in a component of type i where $1 \le i \le k$ and k is the number of places and transitions. In SRE, component types 1 through 6 correspond to PE, IOL, S, BL, HL, SBL and $k = 6$, respectively.

Although the SPN in Fig. 2, might provide the number of operational configurations required, it fails to consider the cases when enough spares are available but reconfiguration cannot take place (due for example to a peculiar distribution of faults). As a consequence, this approach might provide overly optimistic reliability estimates. Conceivably, a different SPN model can be used to accurately represent the dependency of successful reconfigurations on fault distributions. However, such a SPN would itself consist of a large number of places and transitions (which increases with the size of the array) that would generate a large state space in the underlying Markov chain. One of the intents of this paper is to provide an extension of the SPN concept so that dependence on fault distribution can be accounted for in a model with a complexity comparable to that of Fig. 2 regardless of the size of the array and reconfiguration scheme used.

B. MSPN Representation

The fact that several types of faults affect the array in the same manner, suggests a more compact SPN-like representation, which is referred to as Modified SPN (MSPN).

Definition: An MSPN is defined as $MSPN = (P, T, A, M_0, Pr, Sq, B, Cv)$ where $P = \{p_1, \cdots, p_k\}$ is the set of places, $T = \{t_1, \cdots, t_s\}$ is the set of transitions, $A \subset \{P \times T\} \bigcup \{T \times P\}$ is the set of input and output arcs, M_0 is an initial marking, Pr is a set of probability functions $P(x|M_q, t_i)$ associated to transitions $t_i \epsilon T$, Sq is a set of sequences S_i of transitions that fire immediately after an exponential firing of transition $t_i \epsilon T$, B is the set of binary transition vectors B_i defined for each $t_i \epsilon T$, Cv is the set of coverage constants c_i associated with each $t_i \epsilon T$. □

An MSPN representation associates a set of attributes with each transition t_i; i.e., $< P(x|M_q, t_i), S_i(M_q), B_i, c_i >$ where:

$P(x|M_q, t_i)$ defines a discrete probability function where x represents a random marking M_j in a set R directly reachable from a particular marking M_q; the notation Pr^i_{qj} is used to denote the *probability of reconfiguration* $P(x = M_j|M_q, t_i)$, i.e., the probability that the net is in marking M_j after t_i fires when the net is in marking M_q;

$S_i(M_q)$ is a sequence of transitions that will fire immediately after t_i fires. If no immediate firing is required then $S_i(M_q)$ is a null sequence. Depending on the reconfiguration scheme, S_i can be identical for all markings or can be determined in terms of M_q;

B_i is a binary transition vector with k elements b_l such that, $b_l = 1$ if the failure of the lth component triggers the transition t_i and $b_l = 0$ otherwise. This vector facilitates the merging of nondistinct markings, the derivation of probability transition vectors (defined in Section IV), and the derivation of flags (i.e., failure rate conditions) that signal a possible nonoccurrence of a transition;

c_i is a coverage factor associated with t_i, such that if t_i is triggered by the failure of a spare (inactive) component then $c_i = 1$; if t_i is triggered by the failure of an active component then c_i corresponds to the probability of detection given that a fault occurs (i.e., $c_i \le 1$).

For each transition t_i in a MSPN, there is a set A_i of I/O places and a set V_i of multiplicities μ associated with each arc in A_i. When the MSPN is in a marking M_q and a transition t_i fires, the number of tokens m_{lq} in each place $p_l \epsilon A_i$ is then modified as follows: $m_{lj} = m_{lq} - \mu_l^I + \mu_l^O$, resulting in the generation of a new marking M_j. Since the input multiplicities μ_l^I and output multiplicities μ_l^O, can be functions of the size of the array and the current marking, they are referred to as *variable multiplicities*; this concept is a natural extension to the usual notion of "multiplicities" in Petri Net theory and it has been used in the development of the Stochastic Petri Net Package (SPNP) [11]. The notion of "immediate" firing was introduced in the Generalized Stochastic Petri Net model [12] with the classification of *timed* and *immediate* transitions. The former fire after a random exponentially distributed enabling time; the latter fire as soon as they are enabled. Similar distinction is made in the Stochastic Activity Network (SAN) models used in developing Metasan [13]; in a SAN model, *instantaneous* activities occur in a negligible amount of time. Immediate firings in a MSPN capture a sequence of structural changes that may occur when the reconfiguration procedure is triggered and spare support is not available. However, these structural changes must be preserved as states with transition rates modified by the distribution $P(x|M_q, t_i)$ (see Fig. 4). By specifying in S_i which transitions fire immediately, there is no need of an explicit representation of immediate transitions for each t_i with additional places to control the firing sequence until a failure marking is reached. It is important to note that the association of $P(x|M_q, t_i)$ allows the simplification of the model and therefore the reduction of the state space. Otherwise, additional places and transitions must be created to model the effect of faulty spares distributed in the array.

As an example, a MSPN representation of SRE given in Fig. 3, describes the fault model of an $n \times n$ array. Since t_1, t_2, and t_5, in Fig. 2, have the same effect on the array, a single transition t_1 is defined in Fig. 3, with a vector $B_1 = [1\ 1\ 0\ 0\ 1\ 0]$ to indicate that either the failure of a PE, IOL or a HL can cause t_1 to fire. Likewise, t_3 and t_4 become t_2 with a vector $B_2 = [0\ 0\ 1\ 1\ 0\ 0]$. The firing of t_2 represents the failure of a BL or a S either of which

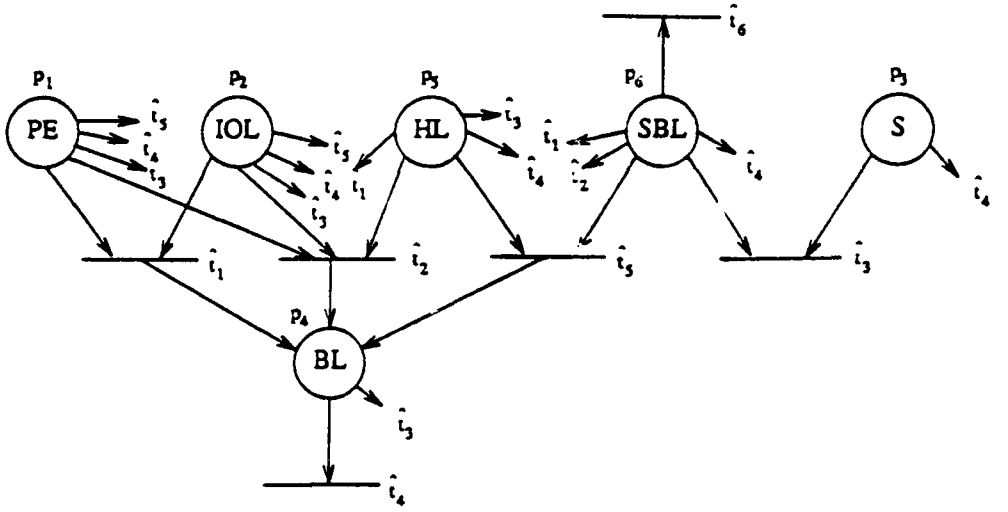


Fig. 2. SPN of the SRE scheme.

is fatal. Transition t_6 becomes t_3 and represents the failure of a SBL with a vector $B_3 = [0\ 0\ 0\ 0\ 0\ 1]$.

Consider now the case when a given operational configuration contains faulty SBL's. The probability that none of them lies in the row that is eliminated when t_1 fires, corresponds to the probability of survival Ps such that $Pr^1_{ij} = Ps$. If at least one SBL is faulty in the row deleted, the array fails to reconfigure with a probability $Pr^1_{if} = 1 - Ps$ (the failure marking is denoted as M_f); i.e., t_2 fires immediatedly and $S_1 = (t_2)$. This follows from the fact that when t_1 fires, the required SBL's become BL's and if one of them is faulty t_2 fires to generate a failure marking. In general, probabilities of survival are complicated functions of the characteristics of each reconfiguration scheme (e.g., replacement rules, hardware requirements, dependencies on fault distributions, etc.) and the size and shape of the array. They must be derived for each scheme and in some cases extensive simulation is required due to the complexity of the combinatorial analysis involved. Examples where suitable expressions have been derived include SRE, ARCE, and DR [5]. For SRE, Ps can be estimated via (5.2) where N corresponds to the number of faulty SBL's present in the current marking M_η. Note that both values Ps and $1 - Ps$ form a discrete distribution which is associated with t_1 as indicated in Fig. 3. When t_2 or t_3 fire exponentially, no immediate transition firing is required and S_2 and S_3 are null sequences. In some applications such as ARCE [5], the sequence S_i is not unique as it is selected depending on the current marking.

The set of multiplicities $V_1 = \{\mu^1_1 = n,\ \mu^1_2 = 2n, \mu^1_5 = n-1,\ \mu^1_6 = n, \mu^1_4 = n\}$, indicates how the I/O places in $A_1 = \{\{p_1, p_2, p_5, p_6\}, \{p_4\}\}$ are affected when t_1 fires; the failure of a S or a BL, will cause the transition t_2 to fire. Since these failures are fatal, all places are affected and $A_2 = \{\{p_1, \cdots, p_6\}, \emptyset\}$ with a set of multiplicities (labeled "all" in Fig. 3) $V_2 = \{\mu^1_1 = \#PE_\eta; \mu^1_2 = \#IOI_\eta; \mu^1_3 = \#S_\eta; \mu^1_4 = \#BL_\eta; \mu^1_5 = \#HL_\eta; \mu^1_6 = \#SBL_\eta\}$, which sets all places to zero to indicate a failure marking. Finally, the failure of a SBL (t_6) affects only SBL's; i.e., $A_3 = \{\{p_6\}, \emptyset\}$ with $V_3 = \{\mu_6 = 1\}$.

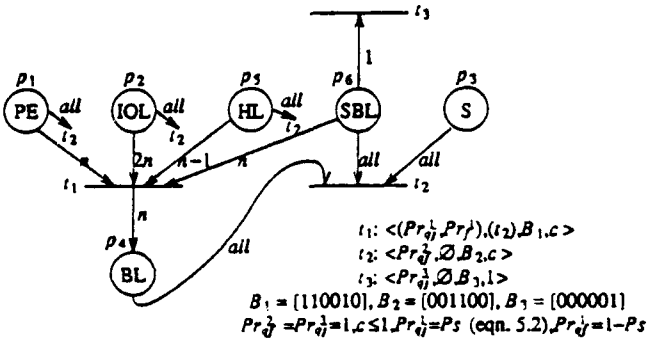


Fig. 3. MSPN of the SRE scheme.

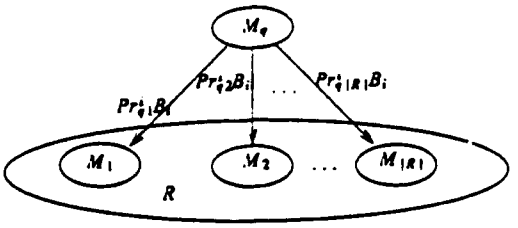


Fig. 4. Marking generation with Pr^1_{ij}.

IV. Reachability Graph

A. Probability Transition Vectors

For each marking M_j generated when t_i fires, B_i and the distribution function $P(x|M_\eta, t_i)$ can be used to generate vectors of the form $Pr^1_{ij}B_i = [pr_1 \cdots pr_k]$ where $pr_l = Pr^1_{ij}$ if $b_l = 1$ and $pr_l = 0$ if $b_l = 0$. These vectors are referred to as the *Probability Transition Vectors (PTV's)*. In the event that two or more nondistinct new markings are generated from the same marking, a merging to a single new marking is carried out by a vector addition of the corresponding probability transition vectors. The use of PTV's is described in Fig. 4. Assuming $j = 1, 2, \cdots, |R|$, then $\sum_{j=1}^{|R|} Pr^1_{ij} = 1$ where R is the set of markings generated by the firing of t_i.

Example 1: Analyzing a particular marking (in a 4×4 array with a MSPN as in Fig. 3) say $M_{18} = [12\ 24\ 20\ 4\ 9\ 11]$, then if t_1 fires (i.e., either a PE, a IOL or a HL failed) the marking $M_{30} = [8\ 16\ 20\ 8\ 6\ 7]$ results with $Pr^1_{18,30} = .66\overline{7}$. Thus, a PTV is given by $Pr^1_{18,30}B_1 = [.66\overline{7}\ .66\overline{7}\ 0\ 0\ .66\overline{7}\ 0]$. However the array may fail with $Pr^1_{18,f} = .333$ due to the existence of one faulty SBL (#PE's - #SBL's = 12 - 11 = 1). Thus, a PTV is given by $Pr^1_{18,f}B_1 = [.333\ .333\ 0\ 0\ .333\ 0]$. Note that $Pr^1_{18,30} = Ps$ and $Pr^1_{18,f} = 1 - Ps$ where Ps is estimated

using (5.2). When t_2 fires (i.e a S or a BL failed), the array fails with probability $Pr^2_{18,f} = 1$, the corresponding PTV is given by: $Pr^2_{18,f}B_2 = [0\ 0\ 1\ 1\ 0\ 0]$. Therefore, the overall PTV associated with the transition to the failure marking when t_1 or t_2 fires is obtained as follows: $Pr^1_{18,f}B_1 + Pr^2_{18,f}B_2 = [.333\ .333\ 1\ 1\ .333\ 0]$. □

B. Derivation of State Transition Rates

The state transition rates a_{ij} between operational states of a continuous-time Markov chain (CTMC) can be expressed in terms of the attributes associated with the transitions in the MSPN, i.e., $a_{qx} = f(P(x|M_q, t_i), B_i, c_i)$. Details on CTMC models and numerical solutions are given in [14].

Let $M_q = [m_{iq}\alpha_i, i = 1 \ldots k]$ where α_i is the failure rate of the ith component and m_{iq} is the number of such components in M_q, then

$$a_{ql} = [\sum_i c_i Pr^i_{ql} b_j;\ b_j \epsilon B_i, j = 1, \ldots, k]\dot{M}^T_q \qquad (4.1)$$

defines the transition rate from state q to an operational state l. The summation is defined over all transitions that fire exponentially and generate the same marking l. It is interesting to note the relationship of (4.1) with the firing rate of a particular transition t_i given by the vector product $B_i\dot{M}^T_q$. Note also that a transition t_i fires only if $B_i\alpha > 0$. A transition to the failure state occurs for lack of support (i.e., enough spares) or lack of coverage. In the first case, lack of support occurs if the reconfiguration algorithm failed due to exhaustion of spare components or the fact that the array fails to reconfigure if a given distribution of faults is not supported by the reconfiguration algorithm. Denote by λ_{qf} the transition to the failure state f for lack of support, then $\lambda_{qf} = [\sum_i Pr^i_{qf} b_j;\ b_j \epsilon B_i, j = 1, \ldots, k]\dot{M}^T_q$. Let λ^c_{qf} be the transition to f for lack of coverage then:

$$\lambda^c_{qf} = \sum_l ([\sum_i (1 - c_i) Pr^i_{ql} b_j;\ b_j \epsilon B_i, j = 1, \ldots, k]\dot{M}^T_q)$$

The overall transition rate to the failure state is: $\tilde{\lambda}_{qf} = \lambda_{qf} + \lambda^c_{qf}$ and the diagonal term of the matrix A is calculated as follows:

$$a_{qq} = -(\sum_{l \neq q} a_{ql} + \tilde{\lambda}_{qf}).$$

Example 2: Consider a particular marking say $M_{18} = [12\ 24\ 20\ 4\ 9\ 11]$, then the following failure rates: $\alpha_0 = 1$, $\alpha_1 = \alpha_2 = \cdots \alpha_k = 0.01$, yield $\dot{M}_{18} = [12\ 0.24\ 0.20\ 0.04\ 0.09\ .11]$. As in example 1, the following probabilities are used: $Pr^1_{18,30} = 0.667$, $Pr^1_{18,f} = 0.333$, $Pr^2_{18,f} = 1$. Let $c_1 = c_2 = c$ then, when t_1 fires the following transition rates in the Markov chain are generated: $a_{18,f} = c[0.667\ 0.667\ 0\ 0\ 0.667\ 0]\dot{M}^T_{18} = 8.2241c$. The transition to the failure state due to lack of support when t_1 and t_2 fire, is $\lambda_{18,f} = ([0.333\ 0.333\ 0\ 0\ 0.333\ 0] + [0\ 0\ 1\ 1\ 0\ 0])\dot{M}^T_{18} = 4.34589$.

For lack of coverage $\lambda^c_{qf} = (1-c)[.667\ .667\ 0\ 0\ .667\ 0]\dot{M}^T_{18} = 8.2241(1-c)$. The overall transition rate to the failure state is given as $\tilde{\lambda}_{18,f} = 8.2241(1-c) + 4.34589$. Considering that the failure of a SBL (t_0) yields a transition rate $a_{18,19} = [0\ 0\ 0\ 0\ 0\ 1]\dot{M}^T_{18} = 0.11$ then the diagonal term is calculated as: $a_{18,18} = -(8.2241c + 4.34589 + 8.2241(1-c) + .11) = -12.68$. □

C. Implementation Procedure

The implementation procedure MODELGEN outlined below merges repeated markings as they are being generated and calculates or modifies the transition rates in the process. The new markings generated every iteration, are targets of the currently visited marking; they are inserted into a linked list of markings which is sorted with respect to the sum of the components indicated by previously set comparison flags. An array of pointers to the newly created targets is updated. A pointer to the current marking is denoted by *cm*. A pointer to the next marking in the linked list is referred to by *next*. A systematic indexing of markings is carried out such that the resulting matrix is always upper triangular; a marking number is assigned to every next marking fetched; a pointer *nxtopr* points to markings which are candidates to be printed or saved in a file. A marking becomes a candidate for output if all its targets have been numbered; therefore, no significant amount of memory is required to generate large models. The procedure stops, when all markings have been fetched from the sorted list. Notice that the linked list corresponds to the reachability graph and as it is formed, both a matrix representation of the Markov model and the reachability graph are printed out or saved as requested by the user.

```
Procedure MODELGEN
Inputs: Set of failure rates (α_i);
File names (to save the matrix representation of the Markov
model);
An MSPN representation of the reconfiguration algorithm.
Outputs: Reachability graph description;
Matrix representation of the Markov model.

Begin
  Set a coverage flag for each transition that is fired by
  nonactive components.
  (this allows for the evaluation of a symbolic matrix for
  different coverage values).
  Set comparison flags (to select those components by
  which the list of markings is sorted).
  Load initial marking
  for each t_i let s_i = B_i α^T
  Let cm point to initial marking
  nxtopr = cm
  while not end of list do
    fetch current marking
    assign a number to current marking
    for each t_i and if s_i > 0 do
      get P(x|M_q, t_i)
      fire t_i and those transitions t_l ∈ S_i
      calculate transition rates
      store targets in temporary table
    end for
    merge repeated targets
    insert new targets in sorted list
    insert pointers to new targets in current marking
    while all targets of nxtopr are numbered do
      output nxtopr marking
      let nxtopr = nxtopr → next
    end while
    let cm = cm → next
  end while
end procedure
```

This procedure has been incorporated into a software package MGRE (Model Generator and Reliability Evaluator) described in [5]. MGRE generates and solves reliability models given the reconfiguration scheme, the size of the array and a set of failure rates.

The execution time of MODELGEN is proportional to the number of states generated and therefore depends on the reconfiguration algorithm. For the applications discussed in [5], the execution time is $O(n^3)$ for $n \times n$ processor arrays.

TABLE II
FAILURE RATES USED FOR THE RESULTS SHOWN IN TABLE III

Array	col.	α_1	α_2	α_3	α_4	α_5	α_6	α_7	Explanation
SRE	a	sx	-	-	-	-	-	-	α_1=pe f.rate
	b	1	0.0	0.0	0.0	0.0	0.0	-	α_2=IOl f. rate
	c	1	0.01	0.0	0.0	0.01	0.0	-	α_3=Switch f.rate
	d	1	0.01	0.01	0.01	0.01	0.01	-	α_4=b.link f.rate
	e	1	0.01	0.1	0.01	0.01	0.01	-	α_5=h.link f.rate
	f	1	0.01	0.01	0.1	0.01	0.1	-	α_6=sp.b.link f.rate
	g	1	0.01	0.01	0.0	0.01	0.0	-	-
ARCE	a	sx	-	-	-	-	-	-	α_3=c.switch f.rate
	b	1	0.0	0.0	0.0	0.0	0.0	0.0	α_4=r.switch f.rate
	c	1	0.01	0.0	0.0	0.01	0.01	0.0	α_5=c.b.link f.rate
	d	1	0.01	0.01	0.01	0.01	0.01	0.01	α_6=r.b.link f.rate
	e	1	0.01	0.1	0.1	0.01	0.01	0.01	α_7=sp.b.l.f.rate
	f	1	0.01	0.01	0.01	0.1	0.1	0.1	sx=simplex

V. PROBABILITY TRANSITION VECTORS IN SRE

For an $n \times n$ processor array using the SRE reconfiguration scheme, the initial marking M_0 of the reachability graph is given as follows: $p_1 : \#PE = n^2$; $p_2 : \#IOL = 2n^2$; $p_3 : \#S = n + n^2$; $p_4 : \#BL = 0$; $p_5 : \#HL = n(n-1)$; $p_6 : \#SBL = n^2$.

A failure marking corresponds to the case when $\#PE = 0$ or $\#SBL < 0$. Transition t_1 will take place if either α_1, α_2, or α_5 is greater than zero; t_2 will take place if either α_3 or α_4 is greater than zero. Likewise, t_3 takes place if $\alpha_6 > 0$. Other applications include ARCE [3] and the Direct Reconfiguration (DR) [4] schemes, reported in [5] and FUSS reported in [15].

To derive PTV's, estimations of probabilities of reconfiguration are required for each reconfiguration scheme. In the SRE case, its MSPN indicates (Fig. 2) that when t_1 fires, two transitions in the Markov model may occur (corresponding to a sequence of two firings in the MSPN: t_1 followed by t_2); one with probability Pr^1_{qj} which corresponds to the probability of survival denoted by Ps. The other transition fires immediately and will lead the processor array to a failure state with probability Pr^1_{qf}. The array will survive with probability Ps if the failure of a PE or an IOL that triggers the reconfiguration algorithm occurs in a row that contains no faulty SBL. Let $\mathcal{N}$ denote the number of faulty SBL in a marking M_q then $\mathcal{N} = \#PE_q - \#SBL_q = m_{1q} - m_{6q}$ where $0 \leq \mathcal{N} \leq n \times n$.

To estimate Ps, it is necessary to find all the possible ways in which $\mathcal{N}$ faulty SBL's can be mapped into a total of $r \times c$ SBL's in the array with a current configuration containing r rows and c columns. Because each row contains c SBL's, up to c faulty SBL's in each row are possible. Let X denote the random number of faulty SBL's in any operational marking, then Ps can be obtained using the following hypergeometric distribution:

$$Ps = Pr(X = x) = \binom{c}{x}\binom{rc-c}{\mathcal{N}-x} \Big/ \binom{rc}{\mathcal{N}}. \quad (5.1)$$

This expression calculates the probability that x faulty SBL's exist in the row to be eliminated during reconfiguration. However, for the case of the SRE reconfiguration scheme, a successful configuration will occur only if $x = 0$, in which case (5.1) is simplified to the following expression:

$$Ps = Pr(X = 0) = \binom{rc-c}{\mathcal{N}} \Big/ \binom{rc}{\mathcal{N}}. \quad (5.2)$$

In [5] probabilities of survival for the ARCE and the DR schemes are derived using also combinatorial expressions.

VI. RELIABILITY ANALYSIS

To illustrate the applicability of MODELGEN, several sets of failure rates (α's) have been selected and described in Table II. For each failure rate set, the Reliability (R), the MTTF, and the Reliability Improvement Factors (RIF) have been computed for a 4×4 processor array. The results obtained are tabulated in Table III and correspond to the analysis of the SRE scheme and the ARCE (Alternate Row Elimination Scheme) scheme analyzed in [5]. In our analysis we have used the PE failure rate as a reference normalized with respect to the time unit such that $\alpha_1 t = 1$ and with a coverage factor of $c = 0.99$. The computations were carried out using the MGRE (Model Generator and Reliability Evaluator) software package described in [5].

The main purpose of the tabulations in Table III is to show the interdependencies of the different components now contained in the model. Column b corresponds to the reliability and MTTF results obtained when only PE's failures are considered. Columns c to f are the results of detailed modeling with different sets of failure rates. Notice for example in column e, the sensitivity to switch failures is reflected in a reduced MTTF. The array is less sensitive to increments in the failure rate of BL's because at all times the number of active switches is greater than the number of active BL's. The simplex sx case in column a, corresponds to the case of the failure of the array when a single processor fails; the Reliability Improvement Factors (RIF's) were calculated with respect to the simplex case.

Let us consider a simplified model such as the one proposed in [16] where the reliability of the array is expressed as $R(t) = R_n(t) \times R_r(t)$. The terms R_n and R_r refer to the reliability of nonredundant and redundant hardware, respectively. For the SRE case, let the number of PE's in the array be the redundant hardware, then

$$R_r(t) = \sum_{i=0}^{n-1} c^i \binom{n}{i} (e^{-n\alpha_1 t})^{n-i} (1 - e^{-n\alpha_1 t})^i.$$

Considering IOL's, HL's, and Switches as the nonredundant hardware, then

$$R_n(t) = e^{-(2n^2\alpha_2 + (n+n^2)\alpha_3 + n(n+1)\alpha_5)t}.$$

Table IV shows the reliability and MTTF results obtained with a set of failure rates as shown in row g in Table II and with $c = 0.99$. The results given by the simplified model show an underestimation of the reliability of a 4×4 array with the SRE scheme as compared to the results obtained by solving the Markov model generated with detailed modeling.

TABLE III
RELIABILITY AND RIF'S FOR SRE AND ARCE WITH $c = 0.99$ AND FAILURE RATES GIVEN IN TABLE II

Array R/RIF	time	a	b	c	d	e	f
SRE	0.1	0.201897	0.97553	0.974196	0.949305	0.792264	0.901154
R	0.3	0.008230	0.74327	0.729796	0.663918	0.3869727	0.489128
	0.5	0.000335	0.428912	0.410413	0.347755	0.141386	0.194080
RIF	0.1	—	32.62	3.93	15.91	3.86	8.50
	0.3	—	3.86	3.67	2.98	1.62	2.04
	0.5	—	1.75	1.70	1.54	1.17	1.27
MTTF	-	-	0.510087	0.496435	0.447789	0.280379	0.33785
ARCE	0.1	0.201897	0.986691	0.986227	0.985946	0.983185	0.985713
R	0.3	0.008230	0.969483	0.968212	0.966729	0.939159	0.959977
	0.5	0.000335	0.95268	0.947997	0.940533	0.831963	0.903652
RIF	0.1	—	59.97	57.95	55.75	46.78	47.29
	0.3	—	32.50	31.20	29.31	16.17	21.72
	0.5	—	21.13	19.22	16.61	5.93	9.65
MTTF	-	-	2.07274	1.90644	1.74707	1.02882	1.28279

TABLE IV
RELIABILITY RESULTS OF SIMPLIFIED AND DETAILED MODELS

Time	Simplified	Detailed
0.1	0.915052	0.954906
0.3	0.613426	0.687296
0.5	0.311454	0.371557
MTTF	0.41571	0.464996

VII. CONCLUSIONS

In this paper Modified Stochastic Petri Nets were proposed as an extension of SPN's to represent fault-tolerant processor arrays and generate detailed Markov models that include several components of the array. A mapping from transitions and markings in a MSPN representation to transitions and states in the Markov model was derived. This mapping allows the construction of the corresponding Markov model as the reachability graph is being generated. The proposed modeling approach has been applied in the analysis of several reconfiguration schemes [5], [15]. The use of this approach was illustrated via the detailed analysis of the SRE reconfiguration scheme. Comparative results show the effect of detailed modeling and component failure interdependencies in the SRE scheme as well as in the ARCE (Alternate Row Column Elimination) scheme analyzed in detail in [5].

Detailed reliability modeling involves three interrelated problems. In the first place a thorough analysis of the array is required such that all interdependencies of failures of different components are defined; this can be a difficult task for complex reconfiguration schemes. Second, the more detail is included in the model, the larger is the state space in the resulting Markov chain. These two problems are inherent to reconfigurable arrays. The use of MSPN's, as input to MGRE, greatly reduces the task of capturing the fault behavior of the array; once the MSPN model is derived it remains invariant as Markov models can be generated for any array size and for any set of failure rates. The last problem involves the solution process of the models generated. Fortunately, large nonstiff models can be solved using the randomization technique [17], which is implemented in MGRE. We emphasize that the models generated are upper triangular sparse matrices which facilitate the solution process. Also, approximation and reduction methods can be applied to large models to derive lower and upper reliability bounds [18]. These techniques are being implemented in MGRE. Finally, considering the fact that reconfiguration algorithms are primarily designed to treat failures of PE's only, a MSPN representation for a given reconfiguration scheme will depend on the assumptions made as to how the algorithm treats failures on component types other than PE's; in this regard, the application chosen (SRE) illustrates the modeling of failure interdependencies as well as the aggregation of probabilities of survival and coverage factors. The combination of marking-dependent variable multiplicities and the set of attributes associated to transitions contribute to make of MSPN's a flexible and compact modeling tool to capture the fault behavior of Fault-Tolerant Processor Arrays. Such flexibility is not found in other packages such as Metasan [13], GreatSPN [19], SHARPE [7], etc. Some of the features of MSPN's are incorporated in SPNP [11] which is being developed independently of MGRE.

REFERENCES

[1] C. S. Raghavendra, A. Avizienis, and M. Ercegovac, "Fault-tolerance in binary tree architectures," *IEEE Trans. Comput.*, vol. C-33, pp. 568–572, June 1984.

[2] I. Koren and D. K. Pradhan, "Modeling the effect of redundancy on yield and performance of VLSI systems," *IEEE Trans. Comput.*, vol. C-36, pp. 344–355, Mar. 1987.

[3] J. A. B. Fortes and C. S. Raghavendra, "Gracefully degradable processor arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 1033–1044, Nov. 1985.

[4] M. G. Sami and R. Stefanelli, "Reconfigurable architectures for VLSI processing arrays," *Proc. IEEE*, vol. 74, pp. 712–722, May 1986.

[5] N. Lopez-Benitez, "Detailed modeling and reliability estimation of fault-tolerant processor arrays," Ph.D. dissertation, Dep. Elec. Eng., Purdue Univ., 1989.

[6] T. E. Mangir and A. Avizienis, "Fault-tolerant design for VLSI effect of interconnect requirements on yield improvement of VLSI designs," *IEEE Trans. Comput.*, vol. C-31, pp. 609–616, July 1982.

[7] R. A. Sahner and K. S. Trivedi, "A hierarchical, combinatorial-Markov method of solving complex reliability models," in *ACM/IEEE Proc. Fall Joint Comput. Conf.*, Nov. 1986, pp. 817–825.

[8] W. G. Bouricious, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *Proc. 24th Nat. Conf. ACM*, Aug. 1969.

[9] W. Reisig, *Petri Nets An Introduction*. Berlin, Germany: Springer-Verlag, 1985.

[10] M. K. Molloy, "Performance analysis using Stochastic Petri Nets," *IEEE Trans. Comput.*, vol. C-31 no. 9, pp. 913–917, Sept. 1982.

[11] G. Ciardo, J. Muppala, and K. S. Trivedi, "SPNP Stochastic Petri Net package," in *Proc. Petri Nets and Perform. Models*, Dec. 1989, pp. 142–151.

[12] M. A. Marsan, G. Conte, and G. Balbo, "A class of generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, pp. 93–122, May 1984.

[13] W. H. Sanders and J. F. Meyer, "METASAN A performability evaluation tool based on Stochastic Activity Networks," in *ACM/IEEE Proc. Fall Joint Comput. Conf.*, Nov. 1986.

[14] A. Reibman and K. S. Trivedi, "Numerical transient analysis of Markov

models." *Comput. and Oper. Res.*, vol. 15, no. 1, pp. 19–36, 1988.
[15] M. Chean and J. A. B. Fortes, "The Full-Use-of-Suitable-Spares (FUSS) approach to hardware reconfiguration for fault-tolerant processor arrays," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 564–571, Apr. 1990.
[16] Y. X. Wang and J. A. B. Fortes, "On the analysis and design of hierarchical fault-tolerant processor arrays," in *Proc. Int. Workshop Defect and Fault Tolerance in VLSI Syst.*," Oct. 1988.
[17] D. Gross and D. R. Miller, "The randomization technique as a modeling tool and solution procedure for transient Markov processes," *Oper. Res.*, vol. 32 no. 2, pp. 343–361, Mar.–Apr. 1984.
[18] M. Smotherman, R. M. Geist, and K. S. Trivedi, "Provably conservative approximations to complex reliability models," *IEEE Trans. Comput.*, vol. C-35, no. 4, pp. 333–338, Apr. 1986.
[19] G. Chiola, "A software package for the analysis of Generalized Stochastic Petri Net models," in *Proc. Int. Workshop Timed Petri Nets*, July 1985.

REFERENCE NO. 10

Wang, Y-X. and Fortes, J.A.B., "On the Analysis and Design of Hierarchical Fault-Tolerant Processor Arrays," International Workshop on Defect and Fault-Tolerance in VLSI Systems, October 6-7, 1988, Springfield, Massachusetts.

Note - This paper addresses the problem of designing hierarchically organized fault-tolerant processor arrays so that their reliability is optimized.

ON THE ANALYSIS AND DESIGN OF HIERARCHICAL FAULT-TOLERANT PROCESSOR ARRAYS

Y-X. Wang and Jose A. B. Fortes*

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

1. INTRODUCTION

The relevance of Processor Arrays (PA's) stems not only from their ability to meet the computational demands of many real-time applications but also from their suitability for efficient implementations using Very Large Scale Integration/Wafer Scale Integration (VLSI/WSI) technology [1]. Fault-tolerant PA's (FTPA's) contain hardware redundancy for some of their components (globally referred to as "redundant hardware") as well as hardware for which spares are not provided (called "non-redundant hardware"). Typically, redundant hardware includes processor elements (PE's) and, sometimes, communication links, whereas non-redundant hardware may include circuitry for control, fault-detection, hardware reconfiguration, etc.. Low reliability in a bad FTPA design may result from, among other reasons, the unreliability of non-redundant hardware. This is particularly true for very large arrays and/or long operation times. This problem can be avoided in well designed hierarchical FTPA's, i.e., processor arrays organized as small fault-tolerant arrays of small FTPA's. This paper addresses the problems of analytically estimating and optimizing the reliability of hierarchical FTPA's.

Extensive work has been done towards devising FTPA's and a non-exhaustive list of references is [2–11]. There exist many fault-tolerant schemes such as the Column Redundancy (CR) ([3] [6] and others), Diogenes (DI) [5], Complex Fault Stealing (CFS) [7] and Triplicated Modular Redundancy (TMR) [10]. Hierarchical FTPA's or closely related approaches have been proposed ([13–16], [20–23] and others) for the yield improvement purposes. Because closed form expressions for the reliability of FTPA's are not easy to derive, one of the motivations of this paper is to provide the general reliability estimation model based on the properties of the hazard function that describes the instantaneous failure rate of FTPA's.

The basic ideas and the approximation model of the hazard function are studied in Section 2. Section 3 introduces two general approximation models for the reliability of FTPA's: the "single-Weibull" and "double-Weibull" approximation models. The first one is used to estimate the reliability of FTPA's where the reliabilities of unit *non-redundant area* (where the non-redundant hardware is laid) and *redundant area* (where the redundant hardware is laid) have the same value. The second one can be used to estimate the reliability of FTPA's where the two reliabilities are independent variables. In Section 4, the motivations of hierarchical structure FTPA are discussed and the methodology for the design of bi-level hierarchical FTPA's is presented. A case study is briefly described that illustrates the very high reliability improvement achievable by a bi-level structure in contrast with the poor reliability of single-level FTPA's with the same amount of redundancy. Section 5 is dedicated to conclusions.

* This research was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-85-k-0588.

2. THE OVERALL RELIABILITY AND HAZARD FUNCTION OF FTPA'S

The overall reliability of FTPA's depends on many factors, e.g., the reconfiguration scheme used, number of redundant components added, size of the array, complexity of PE's, reliability of switches and control circuits, etc. One reconfiguration scheme may have better performance than the others under some circumstances and have worse performance otherwise. For example, as shown in Figure 1, the DI scheme is better than CFS scheme if the size of FTPA's is small whereas for large size FTPA's, the opposite is true (see [13] for details and assumed design and technology parameters).

Without loss of generality, let's assume that an FTPA contains $(n\times n) + k$ PE's, where k is the number of spare PE's. Let $A_n(n,k)$ and $A_r(n,k)$ represent the area of non-redundant and redundant hardware in FTPA's, and $R_r(n,k,r_r)$ and $R_n(n,k,r_n)$ denote the reliabilities of the redundant and non-redundant hardware respectively, where r_r is the reliability of a unit of redundant area (in this paper, the area for a single PE is defined as the unit of area), and r_n is the reliability of a unit of non-redundant area. The reliability of an FTPA can be written as

$$R(n,k,r_n,r_r) = R_n(n,k,r_n) \cdot R_r(n,k,r_r) \ . \tag{2.1}$$

Clearly, the overall reliability of an FTPA is at most as good as the lowest of the reliabilities of the redundant hardware and the non-redundant hardware.

In an FTPA, if some PE's fail, the reconfiguration scheme will try to replace these faulty PE's and make the whole system operational. For a given FTPA with $(n\times n)+k$ PE's, let r_r be the reliability of each PE, then the overall reliability can be written as

$$R(n,k,r_n,r_r) = r_n^{A_n(n,k)} \cdot \sum_{i=0}^{k} s_i \binom{n^2+k}{i} r_r^{(n^2+k-i)}(1-r_r)^i, \tag{2.2}$$

where s_i is the probability that the reconfiguration scheme succeds when there are i faulty PE's in the array. Expressions (2.1) and (2.2) are valid for any type of FTPA. However, in order to facilitate the presentation and discussion of the ideas and results, two assumptions are introduced next and apply to the remainder of this paper. For a given FTPA, it is assumed that, (1) all PE's are redundant hardware and the remaining circuitry corresponds to non-redundant hardware; (2) at the exception of the mechanisms and scheme used for reconfiguration, all other procedures and hardware required for fault tolerance are perfect (e.g., fault detection and location).

In general, the Cumulative Distribution Function (CDF) of any FTPA component is assumed to be exponential with operation time. Then the corresponding hazard function of each component is a constant [12]. In a simplex processor array (the array without any redundancy), a failure in any component causes the whole array to fail and the hazard function of the entire array, h(t), is also a constant, i.e., $h(t)=\lambda>0$. The corresponding reliability is $R(t)=e^{-\lambda t}$.

In FTPA's, a faulty redundant hardware component (e.g. a PE) may be replaced if there is a spare available in the array. Clearly, at time t=0, all components (including the spare ones) are assumed to be good, the probability of a successful reconfiguration is high, and the total number of mortal failures of the array per unit time is low. However, as time goes on, i.e., when the value of t becomes large, some of the spare components are exhausted. The possibility of recovery from additional failures becomes smaller, i.e., the probability of successful reconfiguration is lower and the value of h(t) will increase. Eventually, when most of the spare components are exhausted, the hazard function h(t) will tend to the constant which is the hazard rate of the array with redundancy exhausted. The value of h(t) depends on many factors and, as for the reliability R(t), it is very difficult to obtain the exact expression of h(t). However, the following simple properties of h(t) can be inferred from the discussions above: i) It should be a monotonic increasing function of time t and tend to a constant. ii) It is a continuous function and changes its value smoothly in range (0, $+\infty$). The well known Weibull estimation model [12 17-18] will be studied and used to approximate the h(t) based on these properties. In the rest of this paper, $R^*(t)$ and $h^*(t)$ denote approximations of $R(t)$ and $h(t)$, respectively.

The two-parameter Weibull estimation model has the form

$$h^*(t) = a\cdot t^b, \quad a, b \in \mathbb{R}, \ a>0 \text{ and } b>0. \tag{2.3}$$

The parameters a and b are called the *scale parameter* and *shape parameter*, respectively.

The corresponding formula for the reliability function is

$$R^*(t) = e^{-\int h(t)\,dt} = e^{-\frac{a}{b+1}\cdot t^{b+1}} \tag{2.4}$$

The expressions for both $h^*(t)$ and $R^*(t)$ are very simple compared to other possible approximations, such as Gamma, Beta and Lognormal distributions [12] [19]. On the other hand, according to our experience and previous work [17-18], the Weibull distribution model is very flexible and capable of providing a rich family of approximations for increasing (and decreasing) hazard functions h(t) by adjusting the scale parameter a and the shape parameter b. Because, for FTPA's, $h(t)$ is a positive monotonic increasing function of t, parameters a and b are always positive numbers.

3. GENERAL MODELS FOR FTPA RELIABILITY

The reliability of FTPA's is not only a function of time but also a function of the size of FTPA's. A generalized reliability evaluation model that captures this dependence on size is studied in this section.

For a simplex PA of size N (i.e., N= number of PE's), the hazard function, in terms of the size, is a constant, i.e., $h(N) = \alpha = \ln\frac{1}{r_p}$. The corresponding reliability is $R(N) = r_p^N = e^{-\alpha\cdot N}$. For FTPA's, instead of a constant, the hazard function is a monotonic increasing function of N. This can be explained by the following two facts. First, *the redundancy factor* $\eta = \frac{N}{N-k}$, decreases significantly with the increasing of N. This may cause a shortage of the spares to replace faulty PE's in a very large array and implies that the probability of the failure of the whole system increases with the increase of N. Secondly, because the size of the non-redundant area A_n, usually, is proportional to the size of the array, the reliability of the non-redundant hardware $R_n = r_n^{A_n}$ decreases with the increase of N.

As mentioned in Section 2, the Weibull model can provide a simple and good approximation for those non-linear monotonic increasing (or decreasing) hazard functions. So, h(N) can be modeled by another Weibull formula, i.e.,

$$h^*(N) = a'\cdot N^{b'},\ a', b' \in \mathbb{R}, \text{ and } a'>0,\ b'>0. \tag{3.1}$$

Without loss of generality, let's assume that the variable N is a continuous variable; then the corresponding reliability in terms of the size N can be written as

$$R^*(N) = e^{-\int h(N)\,dN} = e^{-\alpha N^{\beta}}. \tag{3.2}$$

where $\alpha = \frac{a'}{b'+1}$ and $\beta = b'+1$.

There is no time variable t in Equation 3.1 which implies the time is fixed in the equation, i.e., $t = t_0$. Assume that, for a given reconfiguration scheme, the parameter b', which describes the sensitivity of $h^*(N)$ to the size N, is independent of the time t; in other words, assumed to be the increasing rate of the hazard rate, $\frac{h^*(N+1)}{h^*(N)}$, for a given fault-tolerant scheme, is independent of time (parameter a' should be a function of t). If parameter b in Equation 2.3 is independent of the size N, then the hazard rate can be approximated by a model with two variables.

$$h^*(N,t) = c\cdot t^{b}\cdot N^{b'},\ c>0,\ b>0,\ b'>0,\ c, b, b' \in \mathbb{R} \tag{3.3}$$

where c is a new parameter. (The error of the two variable approximation model may increase if parameters b and b' are not independent of N and t, respectively.) The corresponding two variable model for the reliability function is

$$R^*(N,t) = e^{-\int\int h^*(N,t)\,dN\,dt} = e^{-\frac{c}{(b+1)(b'+1)}\cdot t^{b+1}\cdot N^{b'+1}} = e^{-\alpha\cdot t^{\beta}\cdot N^{\gamma}} \tag{3.4}$$

where parameters $\alpha = \frac{c}{(b+1)(b'+1)}$, $\beta = b+1$ and $\gamma = b'+1$ are positive real numbers.

Besides the time t and the size N, the reliability of FTPA also strongly depends on r_p, the reliability of single PE which is determined by the complexity of PE, thchnology etc.. Because $r_p(t) = e^{-\lambda t}$, where λ is the failure rate of a single PE, it follows that $t = \frac{-1}{\lambda}\ln r_p(t)$.

Let $\delta=\frac{1}{\lambda}>0$. By bringing t into Equation 3.4, the reliabilityt then can be written in terms of N, and r_r as

$$R^*(N,r_r(t)) = e^{-\alpha\cdot\delta^\beta\cdot(-\ln r_r(t))^\beta\cdot N^\gamma}. \tag{3.5}$$

Instead of a function of time t, r_r itself can be an independent variable. Letting $\alpha'=\alpha\cdot\delta^\beta$, Equation 3.5 can be rewritten as

$$R^*(N,r_r) = e^{-\alpha'\cdot(-\ln r_r)^\beta\cdot N^\gamma} \tag{3.6}$$

In general, if the reliability of unit non-redundant area always has the same value as the reliability of a single PE, i.e., $r_n=r_r$, then the reliability of the array can be described by a *single-Weibull distribution formula* in terms of N and r_r as in Equation 3.6. However, in some cases, e.g., in hierarchical fault-tolerant structures [14–16], r_n and r_r may vary independently (details are shown in Section 4). Therefore, the reliability of redundant hardware and non-redundant hardware should be described separately in terms of variables r_r and r_n. Because the overall reliability of the system is the product of the reliability of the non-redundant hardware and the redundant hardware, another model, *double-Weibull distribution formula,* is used to estimate the reliability,

$$R^*(N,r_n,r_r)=R_n^*(N,r_n)\cdot R_r^*(N,r_r)=e^{-\alpha_n(-\ln r_n)^{\beta_n}N^{\gamma_n}}\cdot e^{-\alpha_r(-\ln r_r)^{\beta_r}N^{\gamma_r}} \tag{3.7}$$

where α_n, β_n, γ_n, α_r, β_r and γ_r are the six parameters that must be estimated for different FTPA's. Equations 3.6 and 3.7 are used in next section to study the different reconfiguration schemes and to derive a methodology to find the optimal structure of a bi-level FTPA.

4. THE ANALYSIS AND DESIGN OF HIERARCHICAL FTPA'S

The basic idea behind the design of a hierarchical FTPA is best explained if the particular case of a *bi-level FTPA* is considered first. A bi-level FTPA consists of a set of fault-tolerant subarrays. In other words, the full array is partitioned into subarrays and can be thought of as an array of subarrays. Both the subarrays and the array of subarrays use some fault-tolerance scheme. The subarrays are hereon denoted as *1st-level FTPA's* and the array of subarrays is referred to as the *2nd-level FTPA*. A 2nd-level FTPA can be thought of as an FTPA where the basic modules are themselves FTPA's and, physically, it is the same as the bi-level FTPA. The extension to multi-level FTPA's can be easily made by realizing that an n-level FTPA consists of an FTPA whose basic modules are (n-1)-level FTPA's which are FTPA's composed of (n-2)-level FTPA's, etc. For convenience of presentation, bi-level arrays are assumed hereon and, unless stated otherwise, the basic ideas and results apply to multi-level arrays as well.

When faults occur in a bi-level array, reconfiguration is first attempted at the 1st-level FTPA's, and, when reconfiguration fails at the 1st-level, then reconfiguration at the 2nd-level FTPA is attempted. The total non-redundant hardware in a bi-level structure consists of the switches, links and control circuits at the 2nd-level. Therefore, bi-level FTPA's can be expected to have better reliability than single-level FTPA's for at least two reasons: (1) the area of non-redundant hardware in bi-level FTPA's can be smaller than that in single-level FTPA's and (2) the size of arrays and the reconfiguration approach used at each level can be chosen so that optimal reliability results, thus avoiding the inevitable reliability degradation that occurs when the size of single-level arrays grows too large.

Having realized the potential benefits of multi-level FTPA's, from an engineering point of view, it is essential to have a systematic and formal approach to the design of these systems so to optimize the reliability of the overall FTPA. This approach is described in the remainder of this paper.

4.1 The optimization of hierarchical structure

Without loss of generality, let the size of the 1st-level and 2nd-level FTPA's be $(n_1\times n_1)+k_1$ and $(n_2\times n_2)+k_2$, respectively. Here, k_1 and k_2 correspond to the number of spare PE's in 1st-level and the number of spare subarrays in 2nd-level, respectively. Also, the number of processors in the bi-level array is $(n\times n)+k$ where $n=n_1\times n_2$ and $k=k_1\,n_2^2+k_2\,(n_1^2+k_1)$. The reliability of the bi-level FTPA is essentially the reliability of the 2nd-level FTPA, i.e., $R_2=R_2\,(n_2,k_2,R_1)$ where R_1 is the reliability of one 1st-level

FTPA, i.e., $R_1 = R_1(n_1,k_1,r_r)$. Assume that k_1 and k_2 can be expressed as the functions of n_1 and n_2 for the given reconfiguration schemes, R_1 and R_2 can be simplified as $R_1 = R_1(n_1,r_r)$ and $R_2 = (n_2,R_1)$. For a given array size n, different values of n_2 significantly affect both R_1 and R_2. A small value of n_1 may yield a higher value of R_1 but does not necessarily provide a high value on R_2 because of the increase of n_2 ($n_2 = n/n_1$). The example of the reliability of a bi-level FTPA against the number of subarrays n_2 is shown in Figure 2. The value of n_2 for which R_2 is optimal is the solution of the equation

$$\frac{dR_2}{dn_2} = \frac{\partial R_2}{\partial n_2} + \frac{\partial R_2}{\partial R_r} \cdot \frac{\partial R_1}{\partial n_2} + \frac{\partial R_2}{\partial k_2} \cdot \frac{\partial k_2}{\partial n_2} = 0 \tag{4.1}$$

Once the solution n_2^* is obtained, n_1^* can be found because n is assumed to be known. Unfortunately, for a bi-level FTPA design, Equation 4.1 is hard to obtain and solve. In order to deal with this problem, a new methodology based on the general single- and double-Weibull approximation models discussed in Section 3 is proposed in which individual models are used to estimate the reliabilities of the first and the second levels.

Without loss of generality, assume that the area of hardware is normalized so that a single PE always occupies one unit of chip area. Let the reliability of hardware in any unit of area have the same value regardless the kind of circuitry in it. For the reconfiguration schemes used at the first level, because the reliability of a basic PE is given as a constant, the only one variable for R_1 is the size of subarrays n_1. The single-Weibull approximation model can be used to estimate $R_1(n_1)$ as

$$R_1^*(n_1) = e^{-\alpha' \cdot n_1^{\beta}} \tag{4.2}$$

where α' and β are real positive constants.

The approximation form R_2^* of the reliability for the second-level as a function of n_2 is more involved than R_1^*. This is because the reliability of one module (subarray) in 2nd-level is $R_1(n_1)$ which is the function of n_1 (it is also a function of n_2 for a fixed n because $n_1 = n/n_2$) instead of a constant (as r_r in the 1st-level). However, the reliability of unit non-redundant area r_n is still a constant. The reliability of redundant and non-redundant area R_r and R_n in 2nd-level will not always vary congruously with n_2. For example, sometimes, a decrease of R_n due to a larger value of n_2 may cause R_r to increase. Because the goal here is to find the trade-off between n_2 and n_1 on which the overall reliability is optimized, the single-Weibull model can not be used here any more and, the reliability R_r and R_n should be estimated separately in order to represent their variations with n_2. According to the general expression of the reliability in Equation 2.1, the reliability in 2nd-level for some reconfiguration scheme can be written as

$$R_2(n_2,r_n,R_1(n_2)) = R_n(n_2,r_n) \cdot R_r(n_2,R_1(n_2)) \tag{4.3}$$

where

$$R_n(n_2,r_n) = r_n^{A_n(n_2)} \tag{4.4}$$

Because r_n is a constant, $R_2(n_2,r_n,R_1(n_2))$ can be written in terms of n_2 and R_1

$$R_r(n_2,R_1(n_2)) = \sum_{i=0}^{k_2} c_i \binom{n_2^2+k_2}{i} R_1^{(n_2^2+k_2-i)}(1-R_1)^i \quad . \tag{4.5}$$

A_n is the non-redundant area in 2nd-level and k_2 is the number of spare subarrays which is, usually, a known function of n_2 for a given reconfiguration scheme. The double-Weibull approximation model then should be used to express the reliability of the reconfiguration schemes used in 2nd-level of FTPA. Because r_n is a constant, R_n can be modeled as

$$R_n^*(n_2) = e^{-a \cdot n_2^b} \tag{4.6}$$

where a and b are positive real number parameters. From (4.4) and (4.6), it is clear that $A_n(n_2)$ is approximated by $\frac{-a}{\ln r_n} \cdot n_2^b$, i.e., the non-redundant area is assumed to grow proportionally with some power of the number of processors in the array.

The approximation form used for $R_r(n_2,R_1)$ has two variables, n_2 and R_1,

$$R_r^*(n_2,R_1) = e^{-(-lnR_1)^u \cdot v \cdot n_2^w} \tag{4.7}$$

where u, v and w are positive real constants. In summary, the double-Weibull approximation form R_2 is

$$R_2^* = e^{-a \cdot n_2^b} \cdot e^{-(-lnR_1)^u \cdot v \cdot n_2^w} \tag{4.8}$$

Because $n_1 = n/n_2$, letting $\alpha = \alpha' \cdot n^\beta$, (4.2) can be rewritten as

$$R_1^* = e^{-\alpha \cdot n_2^{-\beta}}. \tag{4.9}$$

Substituting R_1^* into (4.8) yields

$$\begin{aligned} R_2^* &= e^{-(a \cdot n_2^b + \alpha^u \cdot n_2^{-\beta u} \cdot v \cdot n_2^w)} \\ &= e^{-(a \cdot n_2^b + \alpha^u \cdot v \cdot n_2^{(w-\beta u)})}. \end{aligned} \tag{4.10}$$

Equation (4.1) can now be solved, i.e.,

$$\frac{dR_2^*}{dn_2} = [a \cdot b \cdot n_2^{b-1} + \alpha^u \cdot v(w - \beta \cdot u) \cdot n_2^{(w-\beta \cdot u-1)}]\ e^{-(a \cdot n_2^b + \alpha^u \cdot v \cdot n_2^{(w-\beta u)})} = 0. \tag{4.11}$$

Because $R_2^* \neq 0$, $\frac{dR_2^*}{dn_2} = 0$ if and only if

$$a \cdot b \cdot n_2^{b-1} + \alpha^u \cdot v(w - \beta \cdot u) \cdot n_2^{(w-\beta \cdot u-1)} = 0. \tag{4.12}$$

Also let's assume $n_2 \neq 0$, (4.12) can be simplified as

$$a \cdot b \cdot n_2^b + \alpha^u \cdot v(w - \beta \cdot u) \cdot n_2^{(w-\beta \cdot u)} = 0. \tag{4.13}$$

Because all parameters are positive real numbers, the necessary and sufficient condition for the existence of a real positive solution to (4.12) is

$$w - \beta \cdot u < 0 \tag{4.14}$$

There are many fault-tolerant schemes (with different values on the parameters u, w and β) which can be used to construct the bi-level structure. If (4.14) is not satisfied for two chosen schemes, then Equation 4.13 has no real number solution and R_2 gets its maximum at one of the boundary points, i.e., n_2=1 or n_2=$n \times n$. This means the bi-level structure can not provide better reliability with these two schemes than a single-level structure with one of these schemes. If this happens, other schemes should be considered and checked by (4.14). If (4.14) holds for two specific schemes, Equation (4.13) can be rewritten as

$$a \cdot b \cdot n_2^{b-(w-\beta \cdot u)} + \alpha^u \cdot v(w - \beta \cdot u) = 0. \tag{4.15}$$

Letting $\theta = w - \beta \cdot u$ and $\phi = -\alpha^u \cdot v \cdot \theta/(a \cdot b)$, then the optimal solution is

$$n_2^* = \phi^{1/(b-\theta)} \tag{4.16}$$

Substituting (4.16) in (4.10) yields the maximum reliability attainable with a bi-level FTPA using two given fault-tolerance schemes

$$R_2^* = e^{-[a \cdot \phi^{b/(b-\theta)} + \alpha^u \cdot d \cdot \phi^\theta/(b-\theta)]}. \tag{4.17}$$

In summary, the optimal hierarchical structure can be obtained by the following four steps: 1) Getting the parameters of Weibull distribution model (via simulation or estimation) for all previous fault-tolerant schemes that are possibly to be used on the hierarchical structure. 2) Choosing different combinations of the schemes in 1st and 2nd level and checking Equation 4.14 to find the possible candidates. 3) Substituting the parameters of the possible combinations of the schemes into (4.17), the one which maximizes R_2^* is the optimal combination of the schemes. 4) Substituting the chosen parameters into (4.16) to achieve the optimal partition n_2 of the hierarchical structure.

4.2 An illustrative example

As an example, the design of a bi-level FTPA with n=36 is considered where the simple CR approach is to be used in the 1st-level and the DI scheme is used in the 2nd-level. For both CR and DI schemes a column of spare processors is used. The area and reliability estimations were computed for both the CR and the DI methods and the reliabilities for each of the levels were approximated as

$$R_1 \simeq e^{-14.88 n_2^{-2.967}} \tag{4.18}$$

$$R_2 \simeq e^{-[0.027 n_2^{1.26} + 0.64 (ln\frac{1}{R_1})^{4.742} n_2^{4.739}]} \tag{4.19}$$

Assume that $r_r = r_s = 0.99$. The values of the variables in (4.16) are $\theta \simeq -11.24$, $b \simeq 1.26$, $\phi \simeq 1.144 \times 10^9$ and the optimal value of n_2 is $n_2^* = 5.3$. Because both n_1^* and n_2^* should be integer numbers with $n_1^* \times n_2^* = 36$, the number $n_2^* = 6$ is used. Then, there are total 1764 PE's in the array and the redundancy factor η is 1.36. The overall reliability is 0.75, where the reliabilities for a single-level array using the DI approach or the CR approach with the same redundancy factor are 0.08 and 0.31, respectively.

5. CONCLUSIONS

Several important related conclusions can be made from the work reported in this paper. First, according to the properties of the hazard rate of the FTPA's, the Weibull estimation model can give a good evaluation on the reliability of different FTPA's. The second conclusion is that, based on reliability estimations, it can be found that non-redundant hardware for fault-tolerant purposes does limit the usefulness of single-level FTPA's beyond a certain size. For different fault-tolerant schemes, different array sizes and different area and technology parameters, there does not exist a scheme for FTPA's which is universally optimal. In other words, different fault-tolerant schemes are optimal for different array sizes and different technologies of VLSI. The third conclusion is that the hierarchical FTPA's do not suffer from the disadvantage pointed out in the second conclusion for single-level FTPA's and can make highly reliable.

To achieve the third conclusion mentioned above, the problem of designing optimal bi-level FTPA's was addressed and a methodology for its solution has been described. The key to this methodology is to avoid
the complexity of the exact analytical expressions by using accurate functional approximations of the reliability of FTPA's at different levels, These approximations are based on Weibull reliability functions.

6. REFERENCES

[1] C. Mead and L. Conway, "Introduction to VLSI Systems," Addison-Wesley, Reading, Mass., 1980.

[2] J. A. Abraham, P. Banerjee, C-Y. Chen, W. K. Fuchs, S-Y. Kuo and A. L. N. Reddy, "Fault Tolerance Techniques for Systolic Arrays," Computer, July 1987, pp. 65-74.

[3] I. Koren and M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant Processor Arrays," IEEE Trans. on Computers, Jan. 1984, pp. 21-27.

[4] S. Y. Kung, "VLSI Array Processors," Prentice-Hall, Englewood Cliffs, N.J. 1987.

[5] A. L. Rosenberg, "The Diogenes Approach to Testable Fault Tolerant Arrays of Processors," IEEE Trans. on Computers, Oct. 1983, pp. 902-910.

[6] J. A. B. Fortes, C. S. Raghavendra, "Gracefully Degradable Processor Arrays," IEEE Trans. on Computers, Nov. 1985, pp., 1033-1044.

[7] M. Sami and R. Steffanelli, "Fault-Tolerance and Functional Reconfiguration in VLSI Arrays," International Conference on Circuits and Systems, 1986.

[8] H. T. Kung and M. S. Lam, "Fault-Tolerance and Two-Level Pipelining in VLSI Systolic Arrays," Proc. Conf. Advanced Research in VLSI, Jan. 1984, pp. 74-83.

[9] J-H. Kim and S.M. Reddy, "A Fault-Tolerant Systolic Array Design Using TMR Method," Proc. IEEE Int'l Conf. Comp. Design: VLSI in Computer, Oct. 1985, pp. 769-773.

[10] J. Von Neumann, "Probabilistic Logics and the Synthesis of Studies," C.E. Shannon and J. McCarthy (Ed.), Priceton University Press, Princeton, New Jersey, 1956.

[11] I. Koren, "Comments on "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors,"" IEEE Trans. on Comp. Vol. C-35, No.1, Jan. 1986, pp. 93.

[12] D.P. Siewiorek and R.S. Swarz, "The Theory and Practice of Reliability System Design," Digital Press, Educational Services, DEC, Bedford, MASS., 1982.

[13] Y. Wang and J.A.B. Fortes, "Hierarchical Approaches to Fault-Tolerace in Processor Arrays," SPIE's 1987 Tech. Symp. on Optics, Electro-Optics, and Sensors, 1987.

[14] T. Ishikawa, S. Momoi, S. Shimada and Y. Ogawa, "Hierarchical Array Processor (HAP) Featuring High Reliability and High System Performance," 1986 Int'l. Conf. on Parallel Processing

[15] M. Wang, M. Cutler and S.Y.H. Su, "On-Line Error Detection and Reconfiguration of Array Processors with Two-Level Redundancy," Proc. Comp. on VLSI and Computers, May, 1987.

[16] N. Tsuda and T. Satoh, "Hierarchical Redundancy for a Linear-Array Sorting Chip," I.F.I.P. international workshop on Wafer-Scale Integration, 23-25 September 1987.

[17] A. Adatia and L.K. Chan, "Robust Estimators of the 3-Parameter Weibull Distribution," IEEE Trans. Reliability, October 1985.

[18] H.L. Harter, A.H. Moore and R.P. Wiegand, "Sequential Tests of Hypotheses for System Reliability Modeled by a 2-Parameter Weibull Distribution," IEEE Trans. Reliability, October 1985.

[19] J.B. McDonald and D.O. Richards, "Hazard Rates and Generalized Beta Distributions," IEEE Trans. Reliability, October 1987.

[20] K.S. Hedlund and L.Snyder, "Systolic Architectures-A Wafer Scale Approach," IEEE 1984 Int'l. Conf. on Comp. Design: VLSI in Computers, 1984, pp. 604-610.

[21] S.Y. Kung, C.W. Chang and C.W. Jen, "On Fault-Tolerance in Array Processors," IEEE 1985 Int'l. Conf. on Comp. Design, pp.764-768 1985.

[22] C. Jesshope and L. Bentley, "Techniques for Implementing two-dimensional wafer-scale processor arrays," IEE Proceedings, Vol. 134, Pt.E, No.2, March 1987, pp. 87-92.

[23] J.H. Hwang and C.S.Raghavendra, "VLSI Implementation of Fault-Tolerant Systolic Arrays," Proc. IEEE Int'l. Conf. on Comp. Design: VLSI in Computers, 1986, pp. 110-113.

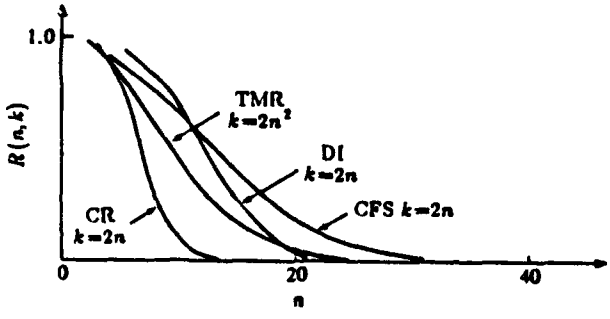


Figure 1. The reliability of different single-level FTPA schemes with (n×n)+k processors; different approaches are optimal for different values of n and low reliability results for large arrays.

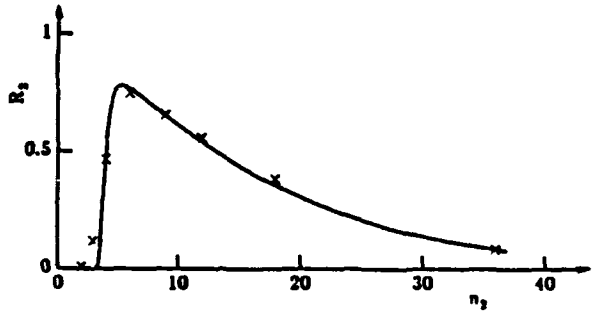


Figure 2. The reliability of a bi-level FTPA, R_n, is the function of the number of the subarrays n_s. The CR scheme is used in low-level and DI scheme is used in high-level with n=36. The values denoted by "x" are obtained by the simulation program and the solid line is according to the approximation model, i.e., Equation 4.10

REFERENCE NO. 13

Wang, Y.-X and Fortes, J.A.B., "Estimates of MTTF and Optimal Number of Spares of Fault-Tolerant Processor Arrays," 20th Int'l Symposium on Fault-Tolerant Computing, June 26-28, 1990, Newcastle Upon Tyne, U.K., pp. 184-191.

Note - This paper presents an efficient analytical approach to the evaluation of Mean-Time-To-Failure (MTTF) of fault-tolerant arrays. It also proposes a design procedure that optimizes MTTF.

ESTIMATES OF MTTF AND OPTIMAL NUMBER OF SPARES OF FAULT-TOLERANT PROCESSOR ARRAYS

Y.X. Wang and Jose A.B. Fortes
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
fortes@ee.ecn.purdue.edu

ABSTRACT

An important problem in the design of Fault-Tolerant Processor Arrays (FTPAs) involves determining the amount of redundancy needed to meet requirements of the Mean Time To Failure (MTTF) or to optimize the MTTF. Given a desired value of MTTF and an FTPA architecture, the *necessary number of spares* (NNS) is defined as the minimum number of spares needed to achieve the desired MTTF. The *optimal number of spares* (ONS) is defined as the number of spares for which the MTTF of the FTPA is maximized. This paper introduces reliability and MTTF models of different FTPAs. Based on these models, approaches which allow for the analytical estimate of the NNS and the ONS are proposed. Knowledge of NNS is suited for FTPAs where non-redundant hardware (hardware for which no redundancy is provided) is considered nearly fault-free. Knowledge of ONS is useful when faults can affect the non-redundant hardware, because in this case overall array reliability may actually decrease when the number of spares increases beyond some value. The quick estimates provided in this paper can be used to help designers in the early phases of design of an FTPA.

1. Introduction

For processor arrays of large size, there is a high probability of failure of one or more processors. This implies that the value of *mean time to failure* (MTTF) may be very small for these processor arrays. In a fault-tolerant processor array (FTPA), spare processors are added and a *fault-tolerant scheme* (or *reconfiguration scheme*) is used to reconfigure the interconnections in order to replace faulty processors when faults occur. Besides the spare processors, some *extra hardware* such as interconnections, switches and control circuits is added to support the reconfiguration. Extensive work has been done towards designing FTPAs and typical schemes such as Column Redundancy (CR) [5], Diogenes (DI) [1], Complex Fault Stealing (CFS), and FUSS [4] are of particular interest in this paper.

In order to design FTPAs, it is necessary to choose one of many possible reconfiguration schemes and to decide how many spares are needed to achieve either maximum MTTF or a prescribed value of it. Detailed design and MTTF evaluation of all possible FTPA architectures are too time-consuming and cannot be used to help a designer make the above decisions. Instead, it is desirable to use valid estimates of MTTF and the number of spares that can be quickly computed for different reconfiguration schemes. Based on these estimates, a designer can choose an initial FTPA architecture that can then be designed and evaluated in detail in order to confirm and refine the estimates. This paper addresses the important questions of how much redundancy is necessary to achieve a required MTTF and what is the optimal redundancy amount that maximizes MTTF.

The cost of an FTPA increases with the number of spares and the extra hardware needed for reconfiguration. Given a desired value of MTTF and an FTPA architecture, the *necessary number of spares* (NNS) is defined as the minimum number of spares needed to achieve that MTTF. If, for a prescribed MTTF, the number of spares is much greater than the necessary number of spares, the spares in excess will render the FTPA unnecessarily expensive. Moreover, because the amount of extra hardware increases with the number of spares, it is not guaranteed that the additional redundancy always improves the MTTF of an FTPA. The decrease of the reliability of the extra hardware may reduce the MTTF significantly, particularly when the size of the array is very large. Thus, it is not always true that the larger the number of spares the higher the MTTF of the array. The *optimal number of spares* (ONS) is defined as the number of spares for which the MTTF of the FTPA is maximized. Different reliability models and MTTF models of FTPAs in terms of the number of spares, survivability of the reconfiguration scheme, reliability of a single processor and other parameters are proposed in this paper in order to analyze and find the NNS and ONS. The results obtained by the proposed estimation models are quite accurate when compared with exact values obtained by extensive computer simulations.

The paper is organized as follows. Basic models of FTPAs and faults are introduced in Section 2. In Section 3, MTTF evaluation models are discussed and the effects of hardware redundancy on MTTF and reliability are compared. The estimation of the necessary number of spares is presented in Section 4. Section 5 discusses and proposes an MTTF model for the FTPAs where failures of interconnections, links, logic control circuits and other hardware elements are taken into consideration. A procedure for estimating the optimal number of spares is proposed in Section 6. Section 7 concludes the work reported in this paper.

2. Reliability Models for FTPAs

For the purposes of this paper, an FTPA with $N' - k$ processors is modeled as $N' + k$ identical modules each of which consists of two parts. One part contains hardware for which redundancy is provided; typically it corresponds to a processing element but, in general, it may also contain part of the hardware used for control, communication and reconfiguration. With this understanding, this part of the module is herein referred to as the *processing element* (PE). The other part of each module contains hardware for which no redundancy is provided. Typically, it includes switches, links and control cir-

This research was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contracts No. 00014-85-k-0588 and No. 00014-88-k-0723.

CH 2877-9/90/0000/0184/$01.00 – 1990 IEEE

cuitry for communication and reconfiguration purposes. In this paper, the hardware of the modules in the array without redundancy is collectively referred to as the *non-redundant hardware* and processing elements are called *redundant hardware* because k spare PEs are provided.

In this paper, a failure of any component is assumed to cause the failure of the entire processor array unless it can be replaced by a functional spare. So, two conditions must be satisfied for an FTPA to be successfully reconfigured when faults occur. First, the reconfiguration scheme must succeed when the given number and pattern of faults are present. The conditional probability of a scheme being able to reconfigure the array given there are i faulty PEs in the array is called *probability of survival* or *survivability* and is denoted s_i. Second, the mechanisms that support reconfiguration must work properly, i.e., the non-redundant hardware must be fault-free.

Faults in an FTPA are assumed to be uniformly distributed and independent. In other words, the failure of a given PE never causes failure of other PEs or non-redundant hardware component and vice versa. Furthermore, it is assumed that failure of a non-redundant component results in failure of the entire processor array. Then the overall reliability of an array can be modeled as

$$R = R_r \cdot R_n \tag{1}$$

where R_r is the reliability of redundant hardware and R_n is the reliability of the non-redundant hardware.

Without loss of generality, let a unit area of the hardware be defined as the area occupied by a single PE. The average area of non-redundant hardware in each module is assumed to be a_n and, therefore, the total area of the non-redundant hardware of the FTPA is $A_n = (N'+k)\,a_n$. Also let the reliability of the hardware in a unit area be $e^{-\lambda t}$ where λ is the *failure rate* of the hardware of a unit area and t is the operation time variable. Then the reliability of the non-redundant hardware is $R_n(t) = e^{-A_n \lambda t}$. If the operation time t is a constant that is *normalized to unity*, R_n also can be written as $R_n = e^{-A_n \lambda}$.

The reliability of the redundant hardware $R_r(t)$ is defined as the summation of the probabilities that the reconfiguration scheme survives i failures for all possible fault distributions and for $i = 0, 1, \ldots, k$, where k is the number of spare PEs. Then,

$$R_r(t) = \sum_{i=0}^{k} s_i \binom{N'+k}{i} \cdot e^{-(N'+k-i)\lambda t}\left(1 - e^{-\lambda t}\right)^i .$$

The combination term in the above equation accounts for all possible fault distributions with i faults. The overall reliability of an FTPA with $N'+k$ PEs can be modeled as

$$R(t) = R_n(t) R_r(t) = e^{-A_n \lambda t} \sum_{i=0}^{k} s_i \binom{N'+k}{i} e^{-(N'+k-i)\lambda t}\left(1 - e^{-\lambda t}\right)^i . \tag{2}$$

Notice that not every failure of the redundant hardware causes the whole array to fail because the faulty redundant hardware may be replaced by spares. The fault which causes the whole array to fail is called the *fatal fault* (which may occur after all spares are exhausted).

For some reconfiguration schemes, such as the Diogenes [1] and FUSS [4] schemes, the survivability s_i is almost constant and close to one. Let S denote the constant value of survivability s_i, then the reliability of this kind of FTPA can be modeled as

$$R(t) = e^{-A_n \lambda t}\, S \sum_{i=0}^{k} \binom{N'+k}{i} e^{-(N'+k-i)\lambda t}\left(1 - e^{-\lambda t}\right)^i . \tag{3}$$

However, for some other reconfiguration schemes, such as the CFS scheme and the CR scheme [2] [5], the survivability s_i is not a constant but a variable in the range $0 \le s_i \le 1$. Because there is no general closed form to express the survivability for this kind of scheme, computer simulations can be used to obtain the survivability s_i [2]. Due to article size limitations, this paper only considers schemes for which the survivability s_i can be approximated by a constant S. The discussion of the cases where the survivability s_i is not constant can be found in [10].

In some previously proposed FTPA designs, it is assumed that the non-redundant hardware is perfect and never fails. This assumption is based on the fact that these non-redundant hardware components are, compared with redundant hardware, smaller and simpler and therefore can be designed carefully and conservatively so as to minimize the probability of operational failure. Therefore, when the amount of the non-redundant hardware is small, the non-redundant hardware can be assumed to be almost fault-free or $R_n(t) \simeq 1$ [1] [2]. Then, the overall reliability of an FTPA can be written as

$$R(t) = S \cdot \sum_{i=0}^{k} \binom{N'+k}{i} \cdot e^{-(N'+k-i)\lambda t}\left(1 - e^{-\lambda t}\right)^i . \tag{4}$$

For large enough processor arrays, non-redundant hardware can not be assumed to be fault-free. The general reliability model of the FTPAs in which $R_n < 1$ is studied later in Section 5.

3. MTTF Model of Processor Arrays

The MTTF should not be confused with the reliability function $R(t)$. As shown in Section 2, the reliability function $R(t)$ is the probability that the array has no fatal faults during the time period $(0, t]$ and is a function of operation time t. However, the MTTF is independent of the operation time t and can also be written as the integration of the reliability of the array as [3]:

$$MTTF = \int_0^{\infty} R(t)\, dt . \tag{5}$$

3.1. Harmonic series model

For FTPAs where the reliability can be modeled by (4), according to (5), the MTTF can be written as

$$\begin{aligned} MTTF &= S \sum_{i=0}^{k} \binom{N'+k}{i} \cdot \int_0^{\infty} e^{-(N'+k-i)\lambda t}\left(1 - e^{-\lambda t}\right)^i dt \\ &= S \sum_{i=0}^{k} \binom{N'+k}{i} \int_0^{\infty} \left(e^{-(N'+k-i)\lambda t} \sum_{j=0}^{i} \binom{i}{j} (-1)^j\, e^{-j\lambda t}\right) dt \\ &= S \sum_{i=0}^{k} \binom{N'+k}{i} \sum_{j=0}^{i} \binom{i}{j} (-1)^j \int_0^{\infty} e^{-(N'+k-i+j)\lambda t}\, dt \\ &= S \sum_{i=0}^{k} \binom{N'+k}{i} \sum_{j=0}^{i} \binom{i}{j} (-1)^j \frac{1}{\lambda(N'+k-i+j)} . \end{aligned}$$

According to [8], the summation in the above equation can be simplified as

$$\sum_{j=0}^{i} \binom{i}{j} (-1)^j \frac{1}{(N'+j)} = \frac{i!}{N'(N'+1)(N'+2)\ldots(N'+i)} .$$

So, the MTTF can be written as

$$MTTF = \frac{S}{\lambda} \sum_{i=0}^{k} \binom{N'+k}{i} \frac{i!}{(N'+k-i)(N'+k-i+1)\ldots(N'+k)} ,$$

$$or \quad MTTF = \frac{S}{\lambda} \sum_{i=0}^{k} \binom{N'+k}{i} \frac{i!(N'+k-i-1)!}{(N'+k)!} .$$

Because $\binom{N'+k}{i} = \frac{(N'+k)!}{(N'+k-i)!i!}$,

$$MTTF = \frac{S}{\lambda} \sum_{i=0}^{k} \frac{(N'+k)!i!(N'+k-i-1)!}{(N'+k-i)!i!(N'+k)!} = \frac{S}{\lambda} \sum_{i=0}^{k} \frac{1}{N'+k-i}$$

or

$$= \frac{S}{\lambda}\left(\frac{1}{N'} + \frac{1}{N'+1} + \frac{1}{N'+2} + ... + \frac{1}{N'+k}\right) \qquad (6)$$

The MTTF model in (6) also can be written as a difference of two Harmonic series as

$$MTTF = \frac{S}{\lambda}\left(\sum_{i=1}^{N'+k} \frac{1}{i} - \sum_{i=1}^{N'-1} \frac{1}{i}\right) = \frac{S}{\lambda}\left(H_{N'+k} - H_{N'-1}\right) \qquad (7)$$

where $H_n = \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$ is the Harmonic series. The MTTF model in (7) is called the *Harmonic series model.*

3.2. Comparison of the models of MTTF and reliability

How the spare PEs improve the MTTF is very different from the way they improve the reliability in large size FTPAs. According to Equation 4, the reliability of an FTPA with N' PEs and k spares is

$$R = S \cdot \left(r^{N'+k} + (N'+k) \cdot r^{N'+k-1} \cdot q + \frac{(N'+k)(N'+k-1)}{2!} \cdot r^{N'+k-2} \cdot q^2 + ... \right) ,$$

where $r = e^{-\lambda \cdot t}$ is the reliability of a single PE and $q = 1 - r$. If it is assumed that $r >> q$, $N' >> k$ and $S \simeq 1$, then the reliability of the array can be approximated as

$$R \simeq r^{N'+k} \cdot \left(1 + \frac{N'q}{r} + \left(\frac{N'q}{r}\right)^2 \frac{1}{2!} + ... \right) .$$

Intuitively, the first item in the above equation is the reliability of the array without any replacement. The second item in the equation can be thought of as the reliability increment for the case when the first faulty PE can be replaced or be recovered. The third item is the reliability increment if the second faulty PE also can be replaced and so on. Let $\Delta_r(i)$ be the increment rate which is the ratio of the reliability increment contributed by the replacement of *ith* faulty PE to the reliability of the corresponding array without any replacement. Figure 1 shows that $\Delta_r(i)$ strongly depends on the failure rate λ and i and it decreases very quickly with the number of spares when λ is small. This implies that a significant reliability improvement is unlikely to result from increasing the number of spare PEs and the reliability cannot be improved constantly by successively adding spares for FTPAs with very small failure rate λ.

However, the MTTF can be improved almost constantly when $N' \gg k$ and the improvement rate contributed by the spare PEs is almost independent of λ and i. Let $\Delta_m(i)$ denote the increment rate which is the ratio of the MTTF increment contributed by the replacement of *ith* faulty PE to the MTTF of the corresponding array without any replacement. Figure 1 shows that, $\Delta_m(i)$ is approximately a linear function of the number i regardless of the value of λ. For example, consider an FTPA where $N' = 100$, $\lambda = 10^{-4}$, $S \simeq 1$ and $k = 1$ (which means only one faulty PE can be replaced), then the MTTF of the array with one spare PE is

$$MTTF_{k=1} = \frac{1}{\lambda}\left(\frac{1}{N'} - \frac{1}{N'+1}\right) = 199 \simeq 2 \cdot MTTF_{k=0} \qquad (8)$$

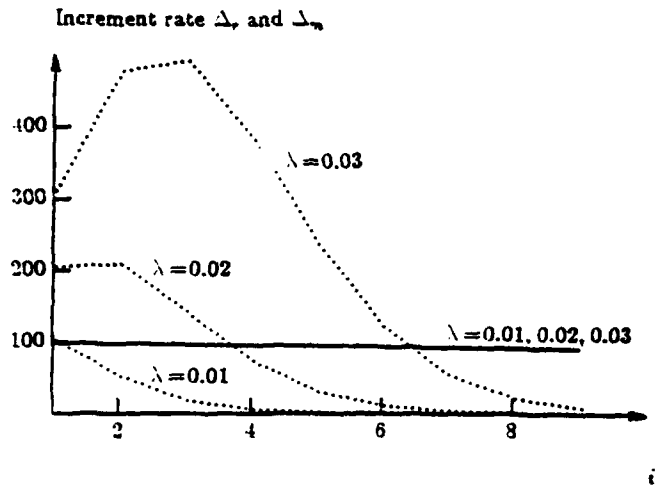


Figure 1: The dotted lines indicate the variation of the increment of reliability $\Delta_r(i)$ with number of spares i. The solid line indicates the variation of the increment of MTTF $\Delta_m(i)$ with i. The logic size of the FTPA's is $N' = 100$ and operation time is $t = 1$. It shows that, $\Delta_r(i)$ is much dependent on the failure rate λ and i, but $\Delta_m(i)$ is almost independent of λ and i.

where $MTTF_{k=0} = 1/(N'\lambda) = 100$ is the mean time to failure of the simplex array without any redundancy. Equation 8 shows that there is almost a 100 percent improvement in the MTTF for the FTPA with only one spare PE. However, for the same FTPA with one spare PE, there is only about 1 percent improvement in the reliability with unity operation time $t = 1$.

In summary, the MTTF can be improved by adding spare PEs when $N' \gg k$ and the improvement rate is independent of the number of spares and the failure rate λ. In contrast, the reliability improvement per added spare PE depends on the number of spare PEs and the failure rate λ.

3.3. Simplified form of MTTF

Because there is no closed form expression for the Harmonic series model of the MTTF,

$$MTTF = \frac{S}{\lambda}\left(\frac{1}{N'} + \frac{1}{N'+1} + \frac{1}{N'+2} + ... + \frac{1}{N'+k}\right) = \frac{S}{\lambda}\left(H_{N'+k} - H_{N'-1}\right) ,$$

further simplification of the MTTF model may be needed in order to find the analytic expression for the number of spares which is required for a given FTPA structure.

According to [8], H_n can be written as another series which may contain an infinite number of items as

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - .....$$

or $H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon$, where $0 < \epsilon < 1/(252\,n^6)$ and $\gamma = 0.5772156649...$ is the *Euler's constant.* Then the MTTF can be expressed as

$$MTTF = \frac{S}{\lambda}\left(\left(\ln(N'+k) + \gamma + \frac{1}{2(N'+k)} - \frac{1}{12(N'+k)^2} - ....\right) - \left(\ln(N'-1) + \gamma + \frac{1}{2(N'-1)} - \frac{1}{12(N'-1)^2} - ....\right)\right) . \qquad (9)$$

The main advantage of this expression, compared to the Har-

monic series model in (7) is that the series in (9) converges much more quickly than that in (7) for those FTPAs having a large number of PEs N'. Assuming the number of PEs in an FTPA is large enough so that only the first three items in series (9) are significant enough to approximate the MTTF and letting $N'-1 \simeq N'$ it follows that

$$MTTF \simeq \frac{S}{\lambda}\left(\ln\frac{N'-k}{N'} - \frac{k}{2N'(N'+k)}\right). \qquad (10)$$

This closed form expression of MTTF is much simpler than the one in (7). The error of this approximation is $O(\frac{1}{N'^2})$ which can be neglected for large size FTPAs. The comparison of these two expressions with different values of N' and k is given in Table 1. The estimation of the necessary number of spares NNS required for an FTPA to achieve a prescribed value of MTTF is discussed in the next section by using the model in (10).

Logic number	Number of spares	MTTF	MTTF*
200	50	227.6443	222.6436
200	100	409.6329	404.6318
200	150	563.5458	558.5443
200	200	696.8987	691.8972
400	50	120.1443	117.6441
400	100	225.3937	222.8936
400	150	320.6131	318.1128
400	200	407.5487	405.0484
600	50	81.6453	79.9786
600	100	155.6984	154.0316
600	150	224.6436	222.9769
600	200	289.1405	287.4738

Table 1: The values of MTTF are obtained from the Harmonic model in (6) and the values marked by MTTF* are obtained from approximation model in (10). The failure rate is $\lambda = 0.001$ and survivability is $S = 1$.

4. Necessary Number of Spares

Given a desired value of MTTF (*prescribed MTTF*) and an FTPA structure (the logic size, fault-tolerant scheme, etc.), the *necessary number of spares* (NNS) is defined as the minimum number of spares needed to achieve the prescribed MTTF. For a prescribed value of MTTF, if the number of spares in an FTPA is much greater than the necessary number of spares, the spares in excess will render the array unnecessarily expensive. In this section, estimates of NNS of FTPA structures with constant survivability are proposed.

Let M_p denote the prescribed value of MTTF of an FTPA. According to the model in (10), the number of spares should satisfy the equation

$$M_p = \frac{S}{\lambda}\left(\left(\ln\frac{N'+k}{N'}\right) - \frac{k}{2N'(N'+k)}\right)$$

or

$$\frac{M_p\cdot\lambda}{S} - \frac{k}{2N'(N'+k)} = \ln\left(N'\frac{+k}{N'}\right). \qquad (11)$$

Let $M = (M_p\cdot\lambda)/S$ and $\delta = k/(2N'(N'+k))$, then Equation 11 can be rewritten as $(N'+k)/N' = e^{M}\cdot e^{\delta}$. Hence the necessary number of spares k for which the array can achieve the prescribed value M_p is

$$k = N'(e^{M}\cdot e^{\delta} - 1). \qquad (12)$$

This is not yet a final solution of the NNS because δ itself is a function of k. In the rest of this section, further simplification and estimation of the solution of Equation 12 are discussed.

To estimate the solution, Equation 12 is studied in three different cases. In the first case, the value of M is small and in the second case, the value of M is large. The third case considers FTPA designs which most frequently happen in practice, and therefore it deserves more attention.

In the first case, it is assumed that, for a given FTPA structure and the prescribed MTTF, the value $M = (M_p\cdot\lambda)/S$ is very small such that $M \le 1/N' \ll 1$. According to (6), the MTTF of an array without redundancy ($k = 0$) is $MTTF = S/(N'\cdot\lambda)$. This implies that a prescribed MTTF of an array is at least $M_p \ge S/(N'\cdot\lambda)$ and it must be equal or greater than the MTTF of the array without any redundancy. In other words, the value $M = (M_p\cdot\lambda)/S$ of any array is at least $M = 1/N'$. Therefore, for a given FTPA structure, when the prescribed MTTF M_p is such that $M = (M_p\cdot\lambda)/S \le 1/N'$, no redundancy is needed for the array and,

$$k = 0 \ . \qquad (13)$$

The second case includes all FTPAs where the prescribed MTTF is large such that $M > 1/N'$. Since $\delta = k/(2N'(N'+k)) < 1/(2N') \ll 1$ usually is very small for large size arrays, it has $e^{\delta} \simeq 1 + \delta$ where the error is $O(\delta^2)$ which is $O(1/(N'^2))$. Then Equation 12 can be approximated as $k = N'(e^{M}\cdot(1+\delta) - 1)$. Because $\delta \ll 1$ and $1 + \delta \simeq 1$, it yields

$$k \simeq N'(e^{M} - 1) \ .$$

Thus, for a given FTPA with $M = (M_p\cdot\lambda)/S > 1/N'$, the necessary number of spares NNS for the array to achieve the prescribed value of MTTF can be estimated as an exponential function as

$$k = \left\lceil N'(e^{(M_p\cdot\lambda)/S} - 1)\right\rceil . \qquad (14)$$

The estimation of the NNS by Equation 14 for the FTPAs with different values of M and N' is shown in Figure 2 by dotted lines. As a comparison, the accurate values of the NNS are indicated by $\times$ in Figure 2. These accurate values are obtained by a computer program in which the calculation of Equation 6 is repeated for different values of $k = 1, 2, \ldots$ until the result of Equation 6 reaches the desired MTTF for a given FTPA. It is clear that Equation 14 provides a good estimation on the NNS for the FTPAs with large values of M.

The first case does not happen very often and can be ignored in large size fault-tolerant processor array designs. This is because if $M = (M_p\cdot\lambda)/S \le 1/N'$, the prescribed MTTF $M_p = S/(N'\cdot\lambda)$ is very low when the logic size of the array N' is large and in practice, the prescribed MTTF is expected to be much larger than this value.

The second case where M is larger than or comparable to one also does not happen very often in practice. This is because for an FTPA structure with given survivability S and failure rate of a single PE λ, a large value of $M = (M_p\cdot\lambda)/S$ may require a huge number of spares which is unreasonably expensive to implement. For example, consider an FTPA with logic size $N' = 400$ and $S = 1$; according to Equation 14, almost 700 spare PEs are required for the array if $M = 1$ is expected. So it may be too expensive to implement an FTPA to achieve the prescribed MTTF which results in a large value of M. Actually, the prescribed MTTF M_p such that $M = (M_p\cdot\lambda)/S$ is very high may not be achievable for large size FTPAs if the failure of non-redundant hardware is considered as indicated later in Section 6. Experience with numerical simulations shows that, for most FTPA designs, the prescribed MTTF M_p is such that $M = (M_p\cdot\lambda)/S$ is in a special range where M is neither as large as comparable to one nor smaller than $1/N'$. Because it applies to

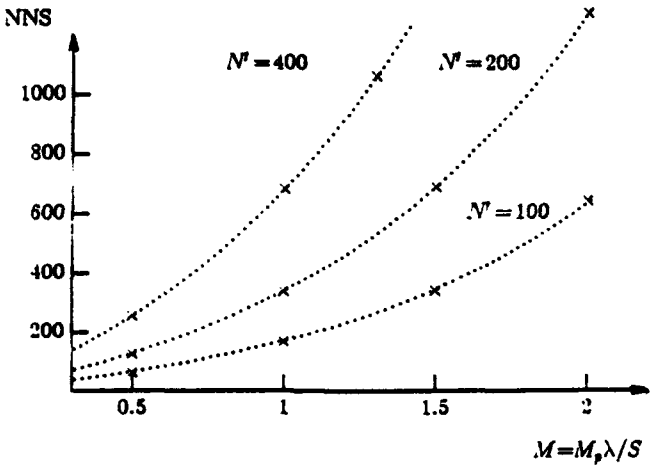


Figure 2: The dotted lines are obtained from exponential estimation model in (14) with different values of N and M. The points marked by × are the accurate values of NNS required for the FTPAs obtained by simulation program according to (6).

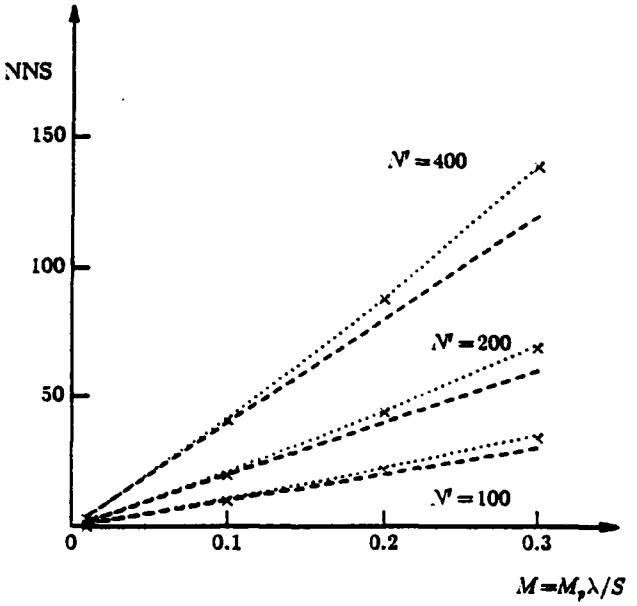


Figure 3: The dashed lines are obtained from linear estimation equation (16) and the dotted lines are obtained from exponential estimation equation (14). The points marked by × are accurate values of NNS of the FTPAs achieved by simulation program according to (6). It shows that, the error of linear estimation equation (16) increases with the value of M of the FTPAs.

most FTPA designs, this special range of M which is a subcase of case two is discussed next as the third case, and a simpler linear equation, instead of (14) can be used to estimate the NNS for this special case.

The third case applies to most FTPAs. Typically in this case, $0.1 > M > 1/N$. Because M is relatively small, the approximation formula $e^x \simeq 1 + x$ for small x can be used for e^M and Equation 14 can be approximated as

$$k = \left\lceil N \left((1 + M) - 1 \right) \right\rceil = \left\lceil N \cdot M \right\rceil . \qquad (15)$$

This provides a very simple linear estimation formula of the NNS for most of the FTPAs

$$k = \left\lceil ((M_p \cdot \lambda)/S) \cdot N \right\rceil . \qquad (16)$$

It shows that, for most FTPAs with given S and λ, the necessary number of spares to achieve the prescribed MTTF is almost proportional to the logic size N and the prescribed MTTF M_p.

Figure 3 compares the estimation models in (16) and (14) to the accurate values of the NNS obtained by Equation 6. The dashed lines in Figure 3 show the estimation of the NNS according to Equation 16, and the dotted lines show the estimation of the NNS obtained by Equation 14. The accurate values of the NNS obtained from the computer program according to Equation 6 are marked by ×. As Figure 3 illustrates, the linear Equation 16 can provide a quite good estimation of the NNS when M is small. Also, (16) is a linear approximation of (14) when M is small. When M is large or $M > 0.1$, estimation model (14) provides better results than the one in (16). The estimation error of model (16) increases with M.

As a summary, for a given prescribed MTTF M_p of an FTPA with nearly perfect non-redundant hardware ($R_n \simeq 1$) and constant survivability S, the NNS can be estimated by three different formulas according to the value of $M = (M_p \cdot \lambda)/S$.

(1) If $M = (M_p \cdot \lambda)/S \leq 1/N$, no spare PEs are needed for the array and $k = 0$.

(2) If $M > 1/N$, the NNS can be estimated by an exponential function as $k = N \cdot (e^M - 1)$.

(3) If $0.1 > M \geq 1/N$, the NNS can be estimated especially by a linear function $k = M \cdot N$.

All estimations of the NNS above are based on the assumption that the non-redundant hardware of FTPA is simple, takes a small area and is almost perfect such that $R_r \simeq 1$. The general case where $R_r < 1$ is discussed next.

5. General Model of MTTF

Because the amount of non-redundant hardware of FTPAs is usually proportional to the size of the array and increases with the number of the PEs in the array, the reliability of non-redundant hardware may decrease drastically and become too low to ignore for very large size FTPAs. As indicated by (1), the overall reliability is the product of the reliabilities of the non-redundant and redundant hardwares. Therefore, for very large size processor arrays, a valid estimation of the desired amount of the redundancy must take the influence of the non-redundant hardware into consideration. In these cases, instead of $R_n \simeq 1$, the expression $R_n = e^{-A_n \lambda t}$ should be used to establish more accurate reliability models for large size FTPAs, where A_n is, as defined in Section 2, the total area taken by non-redundant hardware. Therefore, for the general case, instead of Equation 4, Equation 3 should be used to study and derive MTTF models, i.e.,

$$R(t) = S \cdot e^{-A_n \lambda t} \sum_{i=0}^{k} \binom{N+k}{i} \cdot e^{-(N+k-i)\lambda t} \left(1 - e^{-\lambda t}\right)^i .$$

According to Equation 5, the general MTTF model can be obtained as

$$MTTF = \int_0^\infty R(t)\, dt$$

$$= S \sum_{i=0}^{k} \binom{N'+k}{i} \cdot \int_0^{\infty} e^{-(N'+k+A_n-i)\lambda t} (1-e^{-\lambda t})^i \, dt$$

$$= S \sum_{i=0}^{k} \binom{N'+k}{i} \cdot \int_0^{\infty} (e^{-(N'+k+A_n-i)\lambda t} \sum_{j=0}^{i} \binom{i}{j} (-1)^j e^{-j\lambda t}) \, dt$$

$$= S \sum_{i=0}^{k} \binom{N'+k}{i} \cdot \sum_{j=0}^{i} \binom{i}{j} (-1)^j \int_0^{\infty} e^{-(N'+k+A_n-i+j)\lambda t} \, dt$$

$$= S \sum_{i=0}^{k} \binom{N'+k}{i} \cdot \sum_{j=0}^{i} \binom{i}{j} (-1)^j \frac{1}{\lambda(N'+k-A_n-i+j)} \qquad (17)$$

Because the summation of the combination in (17) can be simplified as $\sum_{j=0}^{i} \binom{i}{j} (-1)^j \frac{1}{(C+j)} = \frac{i!}{C(C+1)(C+2)...(C+i)}$ [8] where C is not necessarily integral, the MTTF of FTPAs can be expressed as

$$\frac{S}{\lambda} \sum_{i=0}^{k} \binom{N'+k}{i} \frac{i!}{(N'+k+A_n-i)(N'+k-A_n-1-i)...(N'+k+A_n)}. \qquad (18)$$

The total area of non-redundant hardware A_n should not be assumed to be very small for large size FTPAs in this section, otherwise the model in Equation 10 can be used to estimate the MTTF as discused in Section 4. To simplify (18), the total area of non-redundant hardware $A_n = (N'+k)a_n$ can be approximated as the nearest integer $B = \lfloor A_n \rfloor \geq 1$. This is because usually N' is large and the relative error between $N'+A_n$ and $N'+B$ is very small. Then the equation above can be simplified as

$$MTTF = \frac{S}{\lambda} \sum_{i=0}^{k} \binom{N'+k}{i} \frac{i!(N'+k+B-1-i)!}{(N'+k+B)!} \, . \qquad (19)$$

Because $\binom{N'+k}{i} = \frac{(N'+k)!}{(N'+k-i)!i!}$,

$$MTTF = \frac{S}{\lambda} \sum_{i=0}^{k} \frac{(N'+k)!i!(N'+k+B-1-i)!}{N'+k-i)!i!(N'+k+B)!}$$

$$= \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!} \sum_{i=0}^{k} \frac{(N'+k+B-1-i)!}{(N'+k-i)!}$$

$$= \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!} \sum_{i=0}^{k} (N'+k+1-i)(N'+k+2-i)....(N'+k+B-1-i)$$

$$= \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!} \sum_{i=0}^{k} (N'+1+i)(N'+2+i)....(N'+B-1+i)$$

$$= \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!} \sum_{i=N'+1}^{N'+k+1} i(i+1)(i-2)....(i+B-2) \qquad (20)$$

Because $B \geq 1$, then

$$MTTF = \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!} \times \left(\sum_{i=1}^{N'+k+1} i(i+1)...(i+B-2) - \sum_{i=1}^{N'} i(i+1)...(i+B-2) \right) \qquad (21)$$

It is easy to prove by induction that $\sum_{i=1}^{N'} i(i+1)...(i+k) = \frac{1}{k+2} \prod_{i=1}^{k} N'(N'+1)...(N'+i+1)$ [8], so Equation 21 can be simplified as

$$MTTF = \frac{S}{\lambda} \frac{(N'+k)!}{(N'+k+B)!B} \left(\prod_{i=1}^{B} (N'+k+i) - \prod_{i=1}^{B} (N'+i-1) \right)$$

$$= \frac{S}{\lambda B} \left(1 - \frac{(N'+k)!}{(N'+k+B)!} \prod_{i=1}^{B} (N'+i-1) \right) \, . \qquad (22)$$

Because the area of non-redundant hardware B used as a denominator in (22) is not required to be an integer, the accurate value A_n can be used to substitute B. Then, the general model of MTTF can be written as

$$MTTF = \frac{S}{\lambda A_n} \left(1 - \prod_{i=1}^{B} \frac{N'-1+i}{N'+k+i} \right) \qquad (23)$$

The error caused by using integer B to replace the total area of non-redundant hardware is negligible because the difference between B and A_n is less than or equal to 0.5.

5. Optimal Number of Spares

As discussed in Section 1, because non-redundant hardware can also fail and the non-redundant hardware increases along with the number of spares, it is not always true that the larger the number of spares, the higher the MTTF of the FTPA. As indicated by (2), when the number of spares increases, the reliability of the redundant hardware increases and the reliability of the non-redundant hardware decreases. At the beginning, the overall reliability increases along with the number of spares and it decreases when the number of spares is beyond some value. Therefore, there is a particular number of spares for which the MTTF is maximized. This section discusses how to find the ONS by using the models proposed in Section 4.

Let $X(k) = \frac{S}{\lambda \cdot A_n} = \frac{S}{\lambda(N'+k)a_n}$ denote the first part of Equation 23 and let $Y(k) = (1 - \prod_{i=1}^{B} \frac{N'-1+i}{N'+k+i})$ be the second part of Equation 23. Then $MTTF = X(k) \cdot Y(k)$. Because $X(k)$ decreases with k and $Y(k)$ increases with k, there is an optimal value of k which can maximize $MTTF(k) = X(k) \cdot Y(k)$ for a given FTPA structure. Figure 4 provides an intuitive idea of how the MTTF of FTPAs varies with the number of spares. The dotted lines describe the MTTF of the arrays with $N' = 400$ and dashed lines correspond to the MTTF of the arrays with $N' = 100$. The survivability of the arrays is $S = 1$. It is clearly shown that the MTTF increases with the number of spares first, and then after some point, it starts to decrease monotonically. For example, consider an array with $a_n = 10^{-1}$ and $N' = 400$; the MTTF is optimized only when the array takes $k = 38$ spare PEs.

It is clear that the optimal number of spares ONS of the FTPA is the minimum number of k which satisfies the following inequality

$$X(k)Y(k) \geq X(k+1)Y(k+1) \, . \qquad (24)$$

This is because before and after the optimal number of spares, the MTTF increases and decreases monotonically, respectively. Unfortunately, it is very difficult to find the accurate solution of the inequality in (24) if a numerical method is not used. Some further simplification of inequality 24 is discussed in the rest of the section based on which a simple estimation of the ONS is obtained.

As indicated in Section 2, the reliability of non-redundant hardware is $R_n = e^{-A_n \lambda t} = e^{-(N'+k)a_n \lambda t}$ and the reliability of a simplex array without any redundancy ($k = 0$) is $R = e^{-N'\lambda t}$. Because the overall reliability of an FTPA is the product of the reliabilities of redundant hardware and non-redundant hardware, i.e., $R = R_r \cdot R_n$, one necessary condition for an FTPA to achieve a better reliability than that of the corresponding simplex array with the same logic number of PEs is that the total area of non-redundant hardware in the FTPA should be much smaller than the total area of the redundant hardware of the array, i.e., $A_n = (N'+k)a_n \ll N'$. If the area of the non-redundant hardware of an FTPA is equal to or comparable with the area of the redundant hardware in the

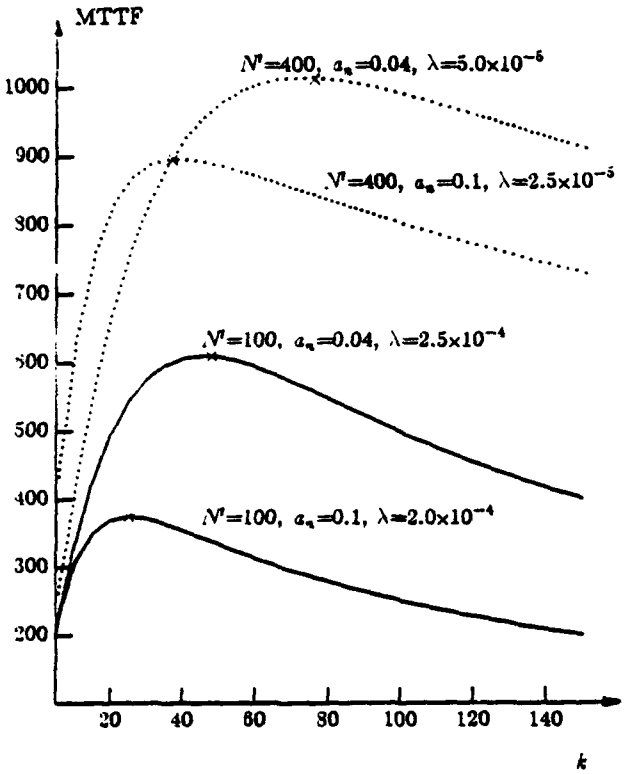


Figure 4: The variations of MTTF of the FTPAs with different values of the parameters are shown by dotted and solid lines. The points marked by x are values of MTTF of the arrays with number of spares estimated by $k = \left\lfloor N'(\sqrt[N'\cdot a_n]{N'\cdot a_n + 1} - 1) - 2 \right\rfloor$.

array, the reliability of that FTPA cannot be better, or it could he even worse than the reliability of the corresponding simplex array no matter what fault-tolerant scheme is used. From these discussions, it can be concluded that one constraint in FTPA design is $a_n \ll 1$. So, it is reasonable to say that in most cases, integers $B(k) = \lfloor (N'+k)a_n \rfloor$ and $B(k+1) = \lfloor (N'+k+1)a_n \rfloor$ have the same value. Then the inequality in (24) can be approximated as

$$\frac{S}{\lambda(N'+k)a_n}\left(1-\prod_{i=1}^{B}\frac{N'-1+i}{N'+k+i}\right) \geq \frac{S}{\lambda(N'+k+1)a_n}\left(1-\prod_{i=1}^{B}\frac{N'-1+i}{N'+k+i+1}\right),$$

and we have

$$1-(N'+k+1)\prod_{i=1}^{B}\frac{N'-1+i}{N'+k+i} \geq -\frac{(N'+k)(N'+k-1)}{N'+k-B+1}\prod_{i=1}^{B}\frac{N'-1+i}{N'+k+i},$$

or

$$\prod_{i=1}^{B}(N'+k+1+i) \geq (B+1)\prod_{i=1}^{B}(N'-1+i). \qquad (25)$$

Notice that

$$\prod_{j=1}^{J}(C-j) = (C+1)(C-2)\ldots(C-J-1)(C+J)$$

can be written as

$$\prod_{j=1}^{J}(C+j) = (C+1)(C+J)\cdot((C+2)(C+J-1))\ldots$$

or

$$\prod_{j=1}^{J}(C+j) = ((C+\tfrac{J+1}{2})^2-(\tfrac{J-1}{2})^2)\times((C+\tfrac{J+1}{2})^2-(\tfrac{J-3}{2})^2)\times (C+\tfrac{J+1}{2})^2-(\tfrac{J-5}{2})^2)\times\cdots$$

If $C \gg (J-1)/2$, it can be approximated by

$$\prod_{j=1}^{J}(C+j) \simeq (C+\tfrac{J+1}{2})^2(C+\tfrac{J+1}{2})^2(C+\tfrac{J+1}{2})^2\ldots \qquad (25a)$$

As mentioned before, the area of the non-redundant hardware is much smaller than the total area for PEs, i.e., $N' \gg B > (B-1)/2$. So, by (25a), inequality (25) can be approximated as

$$(N'+k+1+((B+1)/2))^B \geq (B+1)(N'-1+((B+1)/2))^B .$$

Then the value of k is

$$k = \left\lfloor (N'+\frac{B-1}{2})(\sqrt[B]{B+1} - 1) - 2 \right\rfloor .$$

Because $N' \gg B$ and $N'+((B-1)/2) \simeq N'$, k can be approximated as

$$k = \left\lfloor N'(\sqrt[B]{B+1} - 1) - 2 \right\rfloor . \qquad (26)$$

If it is assumed that $N' \gg k$ so that $B \simeq N'a_n$, then a simpler estimation of the ONS is

$$k = \left\lfloor N'(\sqrt[N'a_n]{N'\cdot a_n+1} - 1) - 2 \right\rfloor . \qquad (27)$$

This formula shows that the value of the ONS of a given FTPA structure is independent of the failure rate of a single PE λ. It also shows that, for the FTPAs which have constant survivability S, the value of ONS is independent of the value of S.

The acceptable accuracy of the estimation of the ONS by (27) is demonstrated in Figures 4 and 5. Figure 4 shows the FTPAs with a relatively small optimal number of spares, and Figure 5 is for the FTPAs with a relatively large optimal number of spares. The dotted and the dashed lines show exactly how the MTTF of the FTPAs varies with the number of spares. These curves are obtained according to Equation 18 by computer programs. The points marked by x are the values of the MTTF corresponding to the estimated optimal number of spares obtained according to (27). Figures 4 and 5 show that the estimation of the ONS by (27) is quite close to the exact value of the optimal number of spares required for different FTPAs and the estimation by (27) is acceptable.

One assumption which makes possible to estimate the NNS and ONS by Equations 14, 16 and 27 is that the survivability of the fault-tolerant scheme can be approximated by a constant S so that (4) provides a good reliability model of the redundant hardware R_r. Some discussions of the cases where the survivability s_i is not close to a constant can be found in [10]. These estimates obtained by Equations 14, 16 and 27 can be readily used by FTPA designers instead of the time-consuming trial-and-error computations of MTTF for many different possible numbers of spares. They may not be as accurate as estimates computed through detailed modeling and sophisticated software tools such as in [6], [7] and others. These tools can capture failure dependencies, the effect of faults of links and control mechanisms and other details, but require time-consuming model construction and computer runs, particularly if different reconfiguration schemes and numbers of

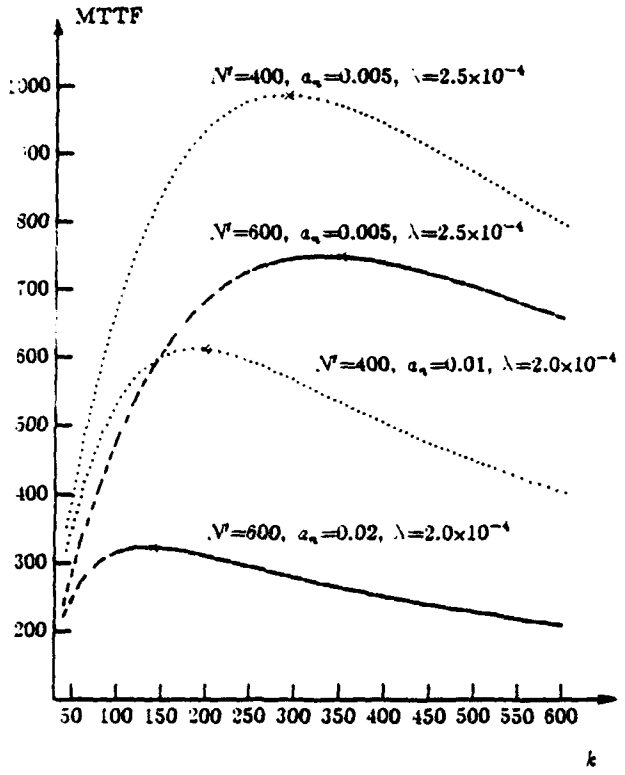


Figure 5: The variations of MTTF of the FTPAs with different values of the parameters are shown by dotted and dashed lines. The points marked by × are values of MTTF of the arrays with the number of spares estimated by $k = \left\lceil N(\sqrt[N \cdot a_q]{N \cdot a_q - 1} - 1) - 2 \right\rceil$.

spares must be considered. Ideally, detailed modeling should be used when a few candidate designs have been identified which are very likely to meet MTTF requirements. The estimates provided in this paper can be used to identify these candidates.

7. Conclusions

An important problem in FTPA design is how to determine the amount of redundancy needed to meet MTTF requirements or to maximize MTTF. This is also important when a designer needs to comparatively evaluate different FTPA reconfiguration schemes. It would be interesting to know the smallest amount of redundancy (the least expensive design) needed to achieve a desired MTTF, or the goal may be to find the best MTTF achievable by any scheme and to know how many spares are required for this best MTTF. Because different technologies and reconfiguration schemes can be chosen to design FTPAs, several cases are considered in this paper. The first case is for the FTPA designs where non-redundant hardware can be assumed to be perfect and never fail and any spare can replace any faulty component regardless of the fault-distribution. In the other case, the failure of non-redundant hardware is taken into consideration. This paper provides different methods that can be effectively used to estimate the necessary number of spares needed to attain a prescribed MTTF and the optimal number of spares which maximize the value of MTTF of a given reconfiguration scheme. Finally, the results of this paper can be readily extended to modular systems other than processor arrays.

References

[1] A. L. Rosenberg, "The Diogenes Approach to Testable Fault Tolerant Arrays of Processors," IEEE Trans. on Computers, Oct. 1983, pp. 902-910.

[2] M. Sami and R. Steffanelli, "Fault-Tolerance and Functional Reconfiguration in VLSI Arrays," Int'l Conf. on Circuits and Systems, 1986.

[3] A. Papoulis, "Probability, Random Variables, and Stochastic Processes," McGraw Hill, New York, N.Y, 1984.

[4] M. Chean, and J.A.B. Fortes, "The Full Use of Suitable Spares (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerant Processor Arrays," IEEE Trans. on Computers, April 1990, pp. 564-571.

[5] A.P. Reeves, "Fault Tolerance in Highly Parallel Mesh Connected Processors," in "Computing Structures for Image Processing," Academic Press. 1983, Chapter 6.

[6] N. Lopez-Benitez, and J.A.B. Fortes, "Detailed Modeling of Fault-Tolerant Processor Arrays," 19th Int'l. Symp. FTCS, June 1989, pp. 545-552.

[7] A.M. Johnson, and M. Malek, "Survey of Software Tools for Evaluation Reliability, Availability, and Serviceability," ACM Computing Surveys, Vol. 20, No.4, Dec. 1988, pp. 227-269.

[8] D.E. Knuth, "The Art of Computer Programming," Vol. 1, and Vol. 2, 1973.

[9] M. Chean, and J.A.B. Fortes, "A Taxonomy for Reconfiguration Techniques for Fault-Tolerant Processor Arrays," Computer, Vol. 23, No. 1, Jan. 1990, pp. 55-69.

[10] Y.X. Wang, "Approximate Estimation of Reliability, Mean-Time-to-Failure and Optimal Redundancy of Fault-Tolerant Processor Arrays," Ph.D thesis, School of Electrical Engineering, Purdue University, W. Lafayette, IN 47907, May 1990.

REFERENCE NO. 15

Shang, W., O'Keefe, M. T., and Fortes, J. A. B., "On Loop Transformations for Generalized Cycle Shrinking," 1991 International Conference on Parallel Processing, August, St. Charles, Illinois, Vol. II, pp. 132-141.

Note - This paper shows how certain compiler optimizations (generically denoted as cycle shrinking transformations) can be unified and generalized by linear schedules such as those used to schedule systolic algorithms.

On Loop Transformations for Generalized Cycle Shrinking

Weijia Shang
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
sw@cacs.usl.edu

Matthew T. O'Keefe
Department of Electrical Engineering
University of Minnesota
Minneapolis, MN55455
okeefe@ee.umn.edu

Jose A. B. Fortes
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
fortes@ee.ecn.purdue.edu

Abstract: This paper describes several loop transformation techniques for extracting parallelism from nested loop structures. One technique is called *selective cycle shrinking* and the other is called *true dependence cycle shrinking*. It is shown how selective shrinking is related to linear scheduling of nested loops and how true dependence shrinking is related to conflict-free mappings of higher dimensional algorithms into lower dimensional algorithms. Methods are proposed in this paper to find the selective and true dependence shrinkings with minimum total execution time by applying the techniques of finding optimal linear schedules and optimal and conflict-free mappings proposed in [13] and [15], respectively.

1. Introduction

This paper describes several loop transformation techniques for extracting parallelism from nested loop structures. This task is often performed by optimizing and parallelizing compilers that have as their goal the transformation and mapping of a serial program into a parallel form that can be executed on a particular architecture [17]. Nested loop structures offer the most fruitful sources of parallelism in serial programs, and it is therefore of paramount importance that the analysis necessary for such parallelization be both precise and efficient.

Algorithms under consideration in this paper are nested loops with regular data dependence structures. Such algorithms can be modeled by a *uniform dependence algorithm* (J, D) [13]. Set J is the *index set* or *iteration space*. Each element in J is an n-tuple integral column vector (called *iteration vector* or *index vector*). Matrix D is the *dependence matrix* where each column is a dependence vector. If computation in one iteration depends on the computation in another iteration, this dependence is represented by the vector difference of the iteration vectors corresponding to these two iterations. All dependences are assumed to be *uniform* which means the dependence relation exists between two iterations as long as the vector difference between these two iterations is equal to the vector representing that dependence relation. This algorithm model can be used for algorithms with nested loops and cyclic dependence structures (*dependence cycles*) described in [11] and is easily related to models used in [5], [10]. Uniform dependence algorithms can be used to model a class of practical algorithms in signal and image processing [4].

In [11], three loop transformation techniques — *simple cycle shrinking*, *selective cycle shrinking* and *true dependence cycle shrinking* — were introduced to transform sequential loops into parallel loops. The concept of cycle shrinking is best illustrated by an algorithm with only one loop. In this case, each dependence vector has only one entry. Cycle shrinking is useful when the minimum of the dependences δ is greater than unity; it transforms a sequential DO loop into two perfectly-nested loops: a sequential outer loop and a parallel inner loop. For example, let a single loop have u iterations, i.e., index set $J = \{1, 2, ..., u\}$; then iterations in sets $\{1, 2, ..., \delta\}$, or $\{\delta+1, ..., 2\delta\}$, ... can be executed in parallel without violating data dependence relations. If the algorithm has a cyclic dependence structure, then the size of the dependence cycle is shrunk by a *reduction factor* δ which is the number of iterations that can be executed in parallel. More examples for the intuitive concepts behind cycle shrinking can be found in [11].

This paper considers generalized selective and true dependence shrinkings.[1] It is shown that the execution order of a generalized selective shrinking can be described by a linear schedule [13]; in addition, it is shown that true dependence shrinking can actually be described by a conflict-free mapping from higher to lower dimensional algorithms [15]. Methods are proposed to find selective and true dependence shrinkings by applying the techniques proposed in [13] and [15]. The resulting selective shrinking is optimal for algorithms with convex polyhedron index sets, while the resulting true dependence shrinking is optimal for algorithms with doubly-nested loops. The resulting transforms outperform those proposed in [11] both in terms of shortest execution time and generality. Also, it is shown that selective shrinking describes wavefront execution [16], [6] and the time-optimal wavefront execution can be obtained by the method discussed in this paper for selective shrinking.

The paper is organized as follows. In Section 2, basic notations, assumptions and ideas and the algorithm model are introduced. Selective and true dependence shrinking are described in Sections 3 and 4, respectively, where the generalized definitions and motivations are given. Methods to find optimal cycle shrinkings are also presented in these two sections. Section 5 concludes this paper and points out some future research.

2. Basic Ideas, Definitions and Motivations

Throughout this paper, *sets*, *matrices* and *row vectors* are denoted by capital letters, *column vectors* are represented by lower case symbols with an overbar and *scalars* correspond to lower case letters. The *transpose* of a vector $\bar{v}$ is denoted $\bar{v}^T$. The vector $\bar{0}$ ($\bar{1}$) denotes the row or column vector whose entries are all zeroes (ones). The dimensions of vector $\bar{0}$ ($\bar{1}$) and whether it denotes a row or column vector are implied

This research was supported in part by the National Science Foundation under Grant DCI-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-88-k-0723.

[1] Both selective and true dependence shrinking always outperform simple shrinking as indicated in [11]. Hence, we do not consider simple shrinking in this paper.

by the context in which they are used. E_i denotes the row vector whose entries are zero except that the *ith* entry is unity. I denotes the identity matrix. The rank of matrix A is denoted *rank(A)*. The set of integers, the set of non-negative integers, the set of positive integers and the set of real numbers are denoted Z, N, N^+ and $\mathbb{R}$, respectively. The empty set is denoted $\emptyset$. The notations $|C|$ and $|\alpha|$ represent the cardinality or the number of elements of set C and the absolute value of scalar α, respectively. Let $\bar{v}$ and $\bar{u}$ be two vectors. Then $\bar{v} \geq \bar{u}$ means every component of $\bar{v}$ is greater than or equal to the corresponding component of $\bar{u}$. Finally, if x is an element of a set S, the notation $x \in S$ is used and this notation is also used to indicate that a column vector $\bar{m}_j$ (or row vector M_i) is a column (row) of a matrix M, i.e., $\bar{m}_j \in M$ ($M_i \in M$) means $\bar{m}_j$ (M_i) is a column (row) vector of matrix M.

This work was motivated by [11] and some of the results reported in this paper are applications of work reported in [13] and [15]. In [11], a special kind of Fortran-like nested loop was considered with a dependence cycle among the statements. This kind of loop has the following form:

```
DO j1 = l1 , u1
 DO j2 = l2 , u2
  ...
   DO jn = ln , un
     S1(J)
     S2(J)                    (2.1)
     ...
     Sp(J)
   END
  ...
 END
END
```

Column vector $\bar{J} = [j_1, j_2, \ldots, j_n]^T$ is the iteration vector. $S_1(\bar{J})$, $S_2(\bar{J})$, ..., $S_p(\bar{J})$ are the p statements in iteration $\bar{J}$. Let $S_i \xrightarrow{\bar{d}} S_k$ denote that statement S_k depends on statement S_i with distance $\bar{d}$. In other words, $S_k(\bar{J})$ depends on $S_i(\bar{J}-\bar{d})$ for all iteration vectors $\bar{J}$ and $\bar{J}-\bar{d}$. A dependence cycle exists in the p statements if $S_1 \xrightarrow{\bar{d}_1} S_2 \xrightarrow{\bar{d}_2} \cdots \xrightarrow{\bar{d}_{p-1}} S_p \xrightarrow{\bar{d}_p} S_1$. Vectors $\bar{d}_i$, $i=1, \ldots, k$ are assumed to be constant and non-zero. The lower and upper bounds of the *ith* nested loop, $1 \leq i \leq n$, are denoted by l_i and u_i, respectively. These bounds were assumed to be constant in [11]. In this paper, it is assumed that l_1 and u_1 are constant and l_i, u_i are linear functions of indices j_k, $1 \leq k \leq i-1$, $k=2, \ldots, n$. Algorithms with n nested loops are called n-dimensional algorithms. A typical example [11] is in Figure 1 (Figure 6a in [11]) where $S_1 \xrightarrow{\bar{d}_1} S_2 \xrightarrow{\bar{d}_2} S_1$ and $\bar{d}_1=[3, 5]^T$ and $\bar{d}_2=[2, 4]^T$. This algorithm is 2-dimensional.

For the purposes of this paper, a pair (J, D) can be used to characterize the algorithm model in (2.1). J is the *index set* or *iteration space*. Each element in J is a n-tuple column vector corresponding to one iteration of the loop. Often in this paper, a column vector in J is called the *index point* or *index vector*. Matrix D is the *dependence matrix* with m columns each of which is a dependence vector $\bar{d}_i$, $i=1, \ldots, m$. For example, the algorithm in Figure 1 can be described by (J, D) where $J = \{\bar{j} : A\bar{j} \leq \bar{b}, \bar{j} \in Z^2\}$ and

```
DO j1 = 3, u1
  DO j2 = 5, u2
       S1: A(j1, j2) = B(j1 - 3, j2 - 5)
       S2: B(j1, j2) = A(j1 - 2, j2 - 4)
  END
END
```

$S_1 \xrightarrow{\bar{d}_1} S_2 \xrightarrow{\bar{d}_2} S_1$, *where* $\bar{d}_1 = \begin{bmatrix}3\\5\end{bmatrix}$, $\bar{d}_2 = \begin{bmatrix}2\\4\end{bmatrix}$

Figure 1: Nested loops with cyclic dependence structure where S_1 depends on S_2 and vice versa.

$$A = \begin{bmatrix}-1 & 0\\ 0 & -1\\ 1 & 0\\ 0 & 1\end{bmatrix}, \bar{b} = \begin{bmatrix}-3\\-5\\u_1\\u_2\end{bmatrix}, \quad D = [\bar{d}_1, \bar{d}_2] = \begin{bmatrix}3 & 2\\5 & 4\end{bmatrix}. \qquad (2.2)$$

The pair (J, D) only captures the structural information of cyclic dependence algorithms in (2.1). Finer-grained information, such as where and when input/output of variables take place, can be ignored for the purposes of this paper. Informally, an index vector in the index set J corresponds to a computation of the algorithm which includes all p statements. Computation $\bar{J}$ (the computation indexed by $\bar{J}$ is often simply described as "computation $\bar{J}$") depends on computation $\bar{J}-\bar{d}_i$, $i=1, \ldots, m$ if $\bar{J}-\bar{d}_i \in J$. Clearly, given a nested loop like the one in (2.1), the pair (J, D) can be constructed. Herein, it is assumed that, the letters n and m always denote the number of nested loops (*algorithm dimension*) and the number of dependence vectors, respectively.

The pair (J, D) can also be used to characterize a more general class of algorithms called *uniform dependence algorithms* whose detailed definition can be found in [13]. The algorithm model in (2.1) can be transformed into a uniform dependence algorithm and all results and techniques presented in this paper apply to both uniform dependence algorithms and the algorithm model in (2.1). Uniform dependence algorithms occur frequently in signal processing and scientific computing applications. Examples include matrix multiplication, LU decomposition, and convolution.

In this paper, only the bounds of the outermost loop need to be constant; the upper and lower bounds of the inner loops can be functions of outer loop indices. This paper considers convex polyhedron index sets. Examples of such index sets include trapezoidal, square and triangular index sets. For 2-dimensional algorithms, these index sets are shown in Figure 2. The convex hulls [1, pp. 35] of these index sets can be described by

$$R = \{\bar{x} : A\bar{x} \leq \bar{b}, A \in Z^{a \times m}, \bar{b} \in Z^a, \bar{x} \in \mathbb{R}^n, a \in N^+\} \qquad (2.3)$$

and

$$J \subseteq \{\bar{j} : \bar{j} \in R \wedge \bar{j} \in Z^n\} \qquad (2.4)$$

where A is called the *index constraint matrix* of J; vector $\bar{b}$ is called the *size vector*; R is called the *convex hull* of index set J. Since R is the convex hull of index set J, extreme points of polyhedron R are always integers and belong to the index set J. Finally, different size vectors correspond to instances of the same algorithm that differ only in their sizes (but have the same shape).

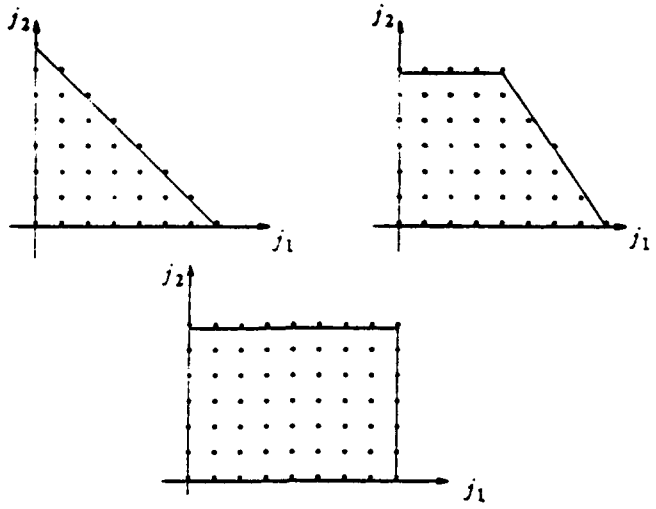
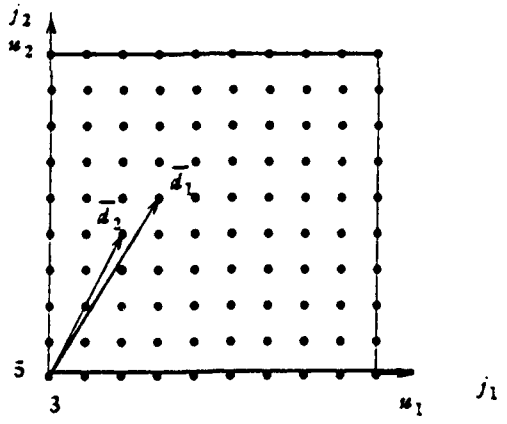
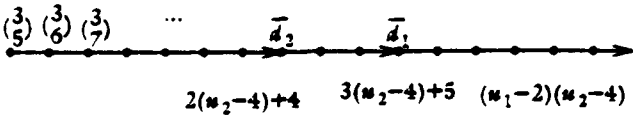


Figure 2: 2-dimensional convex polyhedra.

Figure 3a shows the index set J of the algorithm in Figure 1. Each point corresponds to a computation or an index point. Dependence relations exist among different computations or points. The convex hull of this index set is simply a rectangle which is described by $R=\{\bar{x}=[x_1, x_2]^T: A\bar{x}\leq\bar{b}, \bar{x}\in I\!R^2\}$. Matrix A and the size vector $\bar{b}$ are defined in Equation 2.2. For different values of u_1 and u_2 of size vector $\bar{b}$, the algorithm has its different instances.



(a) The index set of the algorithm in Figure 1.



(b) New 1-dimensional index set after true dependence shrinking.

Figure 3: True dependence shrinking.

3. Generalized Selective Shrinking

In this section, selective cycle shrinking, introduced in [11], is described informally in order to motivate *generalized selective shrinking*, which employs a linear transformation to order algorithm execution. A method for determining optimal generalized selective shrinking is presented, as well as techniques for transforming the original source code according to optimal shrinking.

3.1. Selective shrinking and generalized selective shrinking

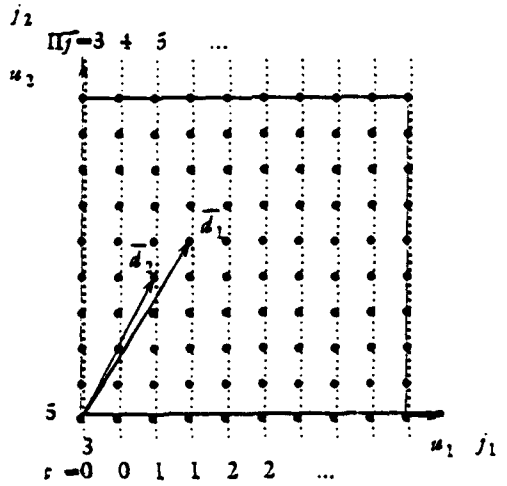
For an n-dimensional algorithm (J, D), let the dependence matrix $D=[d_{ij}]$, $i=1, ..., n$ and $j=1, ..., m$ and $\delta_i=\min\{d_{ij}: j=1, ..., m\}$, $i=1, ..., n$. According to [11], selective shrinking starts from the outermost loop, i.e., at $i=1$ and considers each δ_i, $i=1, ..., n$ in turn. When the first positive number δ_i is met, the process stops. All loops nested inside the *ith* loop are transformed into parallel DOALL loops which means all iterations inside the loop can be executed in parallel. The $i-1$ outermost loops stay unchanged and the *ith* loop is sequential with a loop increment δ_i.

For example, consider the algorithm in Figure 1 where $D=\begin{bmatrix}3 & 2\\ 5 & 4\end{bmatrix}$ and index set J is shown in Figure 3a. This algorithm has a dependence cycle because statement S_1 depends on S_2 and S_2 depends on S_1. Clearly, $\delta_1=\min\{3, 2\}=2$ and $\delta_2=\min\{5, 4\}=4$. Because $\delta_1=2>0$, the outermost loop is blocked and is executed sequentially with a loop increment $\delta_1=2$ and the innermost loop becomes a parallel loop DOALL. The parallel code transformed by the selective shrinking is shown in Figure 4a. With this parallel code, if there are enough processors available, index points $[3, j_2]^T$, $[4, j_2]^T$, $j_2=5, ..., u_2$ can be executed in parallel at time step 0, and index points $[5, j_2]^T$, $[6, j_2]^T$, $j_2=5, ..., u_2$ can be executed in parallel at time step 1, *etc.* The *reduction factor* is $\delta=2(u_2-4)$ which equals the number of computations executed in parallel at one time step. Pictorially, if a set of hyperplanes (lines in this two-dimensional algorithm) $[1, 0]\bar{j}=c$, $c=3, ..., u_1$ are drawn in the index set as shown in Figure 4b, then all the points lying on two consecutive hyperplanes $[1, 0]\bar{j}=c$ and $[1, 0]\bar{j}=c+1$ (e.g., $[1, 0]\bar{j}=3$ and $[1, 0]\bar{j}=4$) can be executed in parallel according to the selective shrinking. This execution order can be described by mapping $\sigma_{[1,0]}(\bar{j})=\left\lfloor\frac{[1, 0]\bar{j}-3}{2}\right\rfloor$ such that computation $\bar{j}$ is executed at time step $\sigma_{[1,0]}(\bar{j})$. Computations indexed by vectors $\bar{j}_1$ and $\bar{j}_2$ can be executed in parallel according to the code in Figure 4a *if and only if* $\sigma_{[1,0]}(\bar{j}_1)=\sigma_{[1,0]}(\bar{j}_2)$. Clearly, this execution order respects the dependence relations. For example, computation $[5, 9]^T$ depends on computation $[3, 5]^T$ by dependence vector $\bar{d}_2$. This dependence relation is respected by selective shrinking because computation $[3, 5]^T$ is scheduled to execute (at $t=0$) before $[5, 9]^T$ (at time $t=1$). In fact, the dependence relation is respected because the inequality $[1, 0]D>0$ is satisfied.

At this point, some interesting observations can be made. First there are other possible mappings σ by which the total execution time may be shorter than the mapping $\sigma_{[1,0]}$. The total execution time by $\sigma_{[1,0]}$ is $\left\lceil\frac{[1, 0]([u_1, 5]^T-[3, 5]^T)+1}{2}\right\rceil=\left\lceil\frac{u_1-2}{2}\right\rceil$. However, if the

```
DO j3 = 3, u1, 2
 DOALL j1 = j3, j3+1
  DOALL j2 = 5, u2
   S1: A(j1, j2) = B(j1 - 3, j2 - 5)
   S2: B(j1, j2) = A(j1 - 2, j2 - 4)
  ENDOALL
 ENDOALL
ENDO
```

(a). Parallel code after selective shrinking.



(b). Pictorial description of the execution order of selective shrinking.

Figure 4: Selective shrinking.

mapping $\sigma_{[0,1]} = \left\lfloor ([0,1]\bar{j}-5)/4 \right\rfloor$ is used, then the total execution time is $\left\lfloor \frac{[0,1]([3,u_2]^T-[3,5]^T)+1}{4} \right\rfloor = \left\lfloor \frac{u_2-4}{4} \right\rfloor$. Note that mapping $\sigma_{[0,1]}$ also respects the dependence relation because $[0,1]D>0$ [13]. Clearly, when $u_2<2u_1$, the total execution time of mapping $\sigma_{[0,1]}$ is shorter than the mapping $\sigma_{[1,0]}$ found using the original selective shrinking technique[2] [11].

A second observation can be made from the example algorithm given in Figure 5, with dependence vector $\bar{d}=[0,1]^T$. In this case, the original selective shrinking technique fails to obtain any speedup because mapping $\sigma_{[1,0]}$, which describes the execution order of the algorithm after selective shrinking, does not respect the dependence relation ($[1,0]\bar{d} = 0$) [13]. However, other feasible mappings with significant reduction factors exist for this algorithm. One possible mapping is $\sigma_{[0,1]} = [0,1]\bar{j}$ with reduction factor $\delta=u_1$. The execution order determined by mapping $\sigma_{[0,1]}$ is pictorially shown in Figure 5 where the execution wavefront is described by dashed hyperplane $[0,1]\bar{j} = c$ and all computations on the same hyperplane $[0,1]\bar{j}=c$ are executed in parallel.

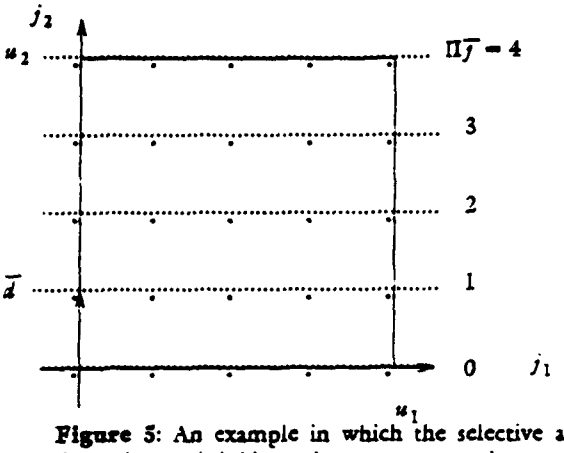


Figure 5: An example in which the selective and true dependence shrinkings do not get speedup or are not feasible.

These two observations indicate that selective shrinking limits the full exploitation of the parallelism in nested loop structures and motivates the definition of generalized selective shrinking as follows.

Definition 3.1 (generalized selective shrinking): For algorithm (J, D) a *generalized selective shrinking* is a mapping $\sigma_\Pi: J \to N$ such that

$$\sigma_\Pi(\bar{j}) = \left\lfloor (\Pi\bar{j} + c)/disp\,\Pi \right\rfloor, \quad \bar{j}\in J \tag{3.1}$$

where $\Pi\in Z^{1\times n}$, $disp\,\Pi=min\{\Pi\bar{d}_i: \bar{d}_i\in D\}>0$, $gcd^3(\pi_1,\ldots,\pi_n)=1$ and $c = -\min\{\Pi\bar{j}: \bar{j}\in J\}$. The row vector Π is called the *schedule vector* specifying σ_Π.

Generalized selective shrinking as defined in Definition 3.1 respects the dependence relations. This means computation $\bar{j}$ is executed only after the execution of computations $\bar{j}-\bar{d}_i\in J$, $i=1, \ldots, m$, upon which computation $\bar{j}$ depends. This follows from the constraint $\Pi D>\bar{0}$. For any algorithm (J, D), the total execution time by the generalized selective shrinking σ_Π is as follows:

$$t = \left\lfloor \frac{\max\{\Pi(\bar{j}_1-\bar{j}_2):\bar{j}_1,\bar{j}_2\in J\}+1}{disp\,\Pi} \right\rfloor = \left\lfloor \frac{\max\{\Pi(\bar{j}_1-\bar{j}_2):\bar{j}_1,\bar{j}_2\in J\}}{disp\,\Pi} \right\rfloor + 1 \tag{3.2}$$

Because total execution time t is minimized *iff* the part inside the floor function is minimized, the problem of finding optimal generalized shrinking can be formulated as follows:

$$\min f = \frac{\max\{\Pi(\bar{j}_1-\bar{j}_2):\bar{j}_1,\bar{j}_2\in J\}}{\min\{\Pi\bar{d}_i:\bar{d}_i\in D\}} \quad \text{subject to} \begin{cases} (1)\ \Pi D>\bar{0} \\ (2)\ gcd(\pi_1,\ldots,\pi_n)=1 \end{cases} \tag{3.3}$$

The optimal solution of the above problem is denoted Π^o. Notice that Π^o is optimal only for the kind of schedule defined in (3.1). It is possible that there exist other schedules

[1] If the values of u_2 or u_1 are not known until run-time, this analysis could be used to generate a run-time test to choose the optimal mapping at run-time based on the relative values of u_2 and u_1. This concept is similar to that proposed in [2] for *multiple version loops*.

[3] $gcd(a_1, \ldots, a_n)$ = the greatest common divisor of $a_1, \ldots, a_n$.

with shorter execution time than Π^e. In [14], a class of algorithms is identified where the optimal linear schedule is optimal for all kinds of schedules. The schedule vector describing the execution order of the selective shrinking defined in [11] for algorithms where $\delta_1 = \min\{d_{11}, \ldots, d_{1m}\} > 0$ is $\Pi = [1, 0, \ldots, 0]$. For the algorithm in Figure 4, the schedule vector for the selective shrinking is $\Pi = [1, 0]$. Because $[1,0][\bar{d}_1, \bar{d}_2] = [3, 2]$, this schedule vector is feasible and respects the dependence relation. However, it is not necessarily the optimal one.

3.2. Finding optimal generalized selective shrinking

In [13], a procedure is proposed to find the optimal schedule vector Π^e for any uniform dependence algorithms with convex polyhedron index sets. To avoid further complicated terminology and definitions, that are needed for a formal presentation of the procedure for any uniform dependence algorithm, only the procedure for uniform dependence algorithms where the loop bounds are constant and the procedure for 2-dimensional algorithms with any convex polyhedron index sets are presented. The procedure for n-dimensional algorithms with any polyhedron index sets and the proofs of the correctness of these procedures can be found in [13].

An index set is called *constant-bounded* if all loop bounds of the associated algorithm are constant, i.e.,

$$J = \{(j_1, \ldots, j_n)^T : l_i \leq j_i \leq u_i, j_i, l_i, u_i \in Z, i = 1, \ldots, n\}. \quad (3.4)$$

Let $D(c_1 \ldots c_x / r_1 \ldots r_y)$ denote the submatrix of D containing the elements in columns $c_1, \ldots, c_x$ and rows $r_1, \ldots, r_y$, i.e., it contains the elements of D at the intersections of columns $c_1, \ldots, c_x$ and rows $r_1, \ldots, r_y$. If $D(c_1 \ldots c_k / r_1 \ldots r_k)$ is nonsingular, an integer row vector $V = [v_1, \ldots, v_k] \in Z^{1 \times k}$ is defined as $V = \beta \bar{1} D^{-1}(c_1 \ldots c_k / r_1 \ldots r_k)$ where β is a positive integer such that $gcd(v_1, \ldots, v_k) = 1$. In other words, V is a vector whose entries are the sums of the corresponding columns of $D^{-1}(c_1 \ldots c_k / r_1 \ldots r_k)$ scaled so that they are integers with the greatest common divisor equal to unity. If $D(c_1 \ldots c_k / r_1 \ldots r_k)$ is nonsingular, then define $\Pi(c_1 \ldots c_k / r_1 \ldots r_k) = VB$, where $B = \begin{bmatrix} E_{r_1} \\ \ldots \\ E_{r_n} \end{bmatrix}$ and E_i is as defined in the first paragraph of Section 2. In words, the subvector $[\pi_{r_1}, \ldots, \pi_{r_n}]$ of $\Pi(c_1 \ldots c_k / r_1 \ldots r_k)$ is the same as V and the remaining entries of $\Pi(c_1 \ldots c_k / r_1 \ldots r_k)$ are zero. Finally, C denotes the set $\{\Pi(c_1 \ldots c_k / r_1 \ldots r_k)$: $1 \leq k \leq n, 1 \leq c_1 < \ldots < c_k \leq m, 1 \leq r_1 < \ldots < r_k \leq n\}$. The following example illustrates the notations and concepts just introduced followed by a theorem which states that the optimal solution Π^e is in C and a procedure which constructs the candidate set C.

Example 3.1: Consider the algorithm (J, D) in Figure 1 where dependence matrix D is as shown in Equation 2.2. According to the definitions just introduced, $D(1/1) = [3]$ (contains the entry in the first column and the first row of D), its corresponding $V = \beta \bar{1} D^{-1}(1/1) = [1]$ where $\beta = 3$, $B = E_1 = [1, 0]$ and $\Pi(1/1) = VB = [1, 0]$; $D(1/2) = [5]$, its corresponding $V = \beta \bar{1} D^{-1}(1/2) = [1]$ where $\beta = 5$, $B = E_2 = [0, 1]$ and $\Pi(1/2) = VB = [0, 1]$; and $D(12/12) = D$, its corresponding $V = \beta \bar{1} D^{-1} = [-1, 1]$ where $\beta = 2$, $B = I$ and $\Pi(12/12) = VB = V = [-1, 1]$. The candidate set $C = \{[1, 0], [0, 1], [-1, 1]\}$. □

Theorem 3.1 [13]: If the index set J of algorithm (J, D) is constant-bounded as defined by (3.4), then the optimal solution Π^e to problem (3.3) belongs to C, i.e.,

$$\Pi^e \in C = \{\Pi(c_1 \ldots c_k / r_1 \ldots r_k): 1 \leq k \leq n, \quad 1 \leq c_1 < \ldots < c_k \leq m, 1 \leq r_1 < \ldots < r_k \leq n\} \quad (3.5)$$

Proof: See [13].

Procedure 3.1 (construction of candidate set C):
Input: Algorithm (J, D) where J is constant-bounded.
Output: A finite candidate set C containing the optimal solution Π^e.
Step 1: $k = 1$, $l = 1$, $C = \varnothing$.
Step 2: $C_k = \varnothing$.
Step 3: Pick an unprocessed combination of k elements from $\{1, \ldots, n\}$. Denote it $\{r_1, \ldots, r_k\}$ where $r_1 < \ldots < r_k$.
Step 4: Pick an unprocessed combination of k elements from $\{1, \ldots, m\}$. Denote it $\{c_1, \ldots, c_k\}$ where $c_1 < \ldots < c_k$.
Step 5: If $D(c_1 \ldots c_k / r_1 \ldots r_k)$ is not singular, then $\Pi_l = \Pi(c_1 \ldots c_k / r_1 \ldots r_k)$ and $C_k = C_k \cup \{\Pi_l\}$, $l = l + 1$.
Step 6: Check if all distinct combinations of k elements from $\{1, \ldots, m\}$ have been processed. If not, go to Step 4.
Step 7: Check if all distinct combinations of k elements from $\{1, \ldots, n\}$ have been processed. If not, go to Step 3.
Step 8: If $k < rank(D)$, then $k = k + 1$, go to Step 2.
Step 9: $C = \bigcup_{k=1}^{rank(D)} C_k$. Stop.

The complexity of Procedure 3.1 is bounded above by $O(\sum_{k=1}^{rank(D)} \binom{n}{k}\binom{m}{k} k^3)$. If $rank(D) = n = m$, then $\sum_{k=1}^{rank(D)} \binom{n}{k}\binom{m}{k} = \sum_{k=1}^{n} \binom{n}{k}^2 \leq (\sum_{k=1}^{n} \binom{n}{k})^2 = (2^n)^2 = 2^{2n}$ and the complexity of Procedure 3.1 is bounded above by $O(2^{2n} n^3)$. As indicated in [13], this upper bounder on the complexity is loose and in practical algorithms, n is small, e.g., $n = 2$ for the algorithm in Figure 1. Next, an example is used to illustrate Procedure 3.1.

Example 3.2: Consider the algorithm in Figure 1. According to Steps 1 and 2 in Procedure 3.1, let $k = 1$, $l = 1$ and $C_1 = \varnothing$. For Steps 3 and 4, there are two possible combinations of one element ($k = 1$) from $\{1, \ldots, n\}$ and two possible combinations of one element from $\{1, \ldots, m\}$ ($n = m = 2$). So in Step 5, $D(1/1)$, $D(1/2)$, $D(2/1)$ and $D(2/2)$ are processed. Each corresponding vector is obtained as follows. $D(1/1) = [3]$ and $\Pi(1/1) = [1, 0]$; $D(2/1) = [2]$, $\Pi(2/1) = [1, 0]$; $D(1/2) = [5]$ and $\Pi(1/2) = [0, 1]$; $D(2/2) = [4]$ and $\Pi(2/2) = [0, 1]$. $C_1 = \{[1, 0], [0, 1]\}$. For $k = 2$, there is only one combination, i.e., $D(12/12) = D$ and $\Pi(12/12) = [-1, 1]$. $C_2 = \{[-1, 1]\}$. So $C = \{[1, 0], [0, 1], [-1, 1]\} = \{\Pi_1, \Pi_2, \Pi_3\}$. Every vector in C is feasible since $\Pi_1 D = [3, 2]$, $\Pi_2 D = [5, 4]$, and $\Pi_3 D = [2, 2]$. The total execution time by each candidate is as follows.

$$t(\Pi_1) = \left\lfloor \frac{[1, 0]([u_1, 5]^T - [3, 5]^T) + 1}{2} \right\rfloor = \left\lfloor \frac{u_1 - 2}{2} \right\rfloor$$

$$t(\Pi_2) = \left\lfloor \frac{[0, 1]([3, u_2]^T - [3, 5]^T) + 1}{4} \right\rfloor = \left\lfloor \frac{u_2 - 4}{4} \right\rfloor$$

$$t(\Pi_3)=\left\lfloor\frac{[-1,1]([3,u_2]^T-[u_1,5]^T)+1}{2}\right\rfloor=\left\lfloor\frac{u_1+2u_2-7}{2}\right\rfloor$$

With little thought, it can be seen that no matter what values u_1 and u_2 take, Π_1 and Π_2 have shorter execution time than Π_3. When $u_2 \leq 2u_1$, vector Π_2 has the minimum execution time and should be the optimal schedule vector and if $u_2 > 2u_1$, Π_1 should be the optimal schedule vector. □

For 2-dimensional algorithms with convex polyhedron index sets (examples of such index sets are shown in Figure 2), let C' denote the candidate set. According to [13], C' contains all vectors Π which (1) are perpendicular to one of the boundary lines of the index set or (2) satisfy equation $\Pi(\bar{d}_i-\bar{d}_k)=0$ where $\bar{d}_i$ and $\bar{d}_k$ are two linearly independent dependence vectors of the algorithm. As defined in Equation 2.3, let A_i and b_i be a row of the index constraint matrix A and an entry of the size vector $\bar{b}$, respectively, then $A_i\bar{j}=b_i$ defines a boundary line of the index set J. Vectors which are perpendicular to that boundary line are described by αA_i where α is a non-zero constant. Thus, C' contains these rows of A each of which is linearly independent of any other rows of A in C'. For example, if A has two rows [1, 0] and [−1, 0], then only one of them is included in C' because they really correspond to the same schedule vector. The following procedure summarizes how to construct the candidate set C' of 2-dimensional algorithms with polyhedron index sets.

Procedure 3.2 (construction of C' for 2-dimensional algorithms):

Input: Index constraint matrix A and dependence matrix D.

Output: A finite candidate set C' containing the optimal solution Π^o.

Step 1: $k=1$, $l=1$, $C'=\emptyset$.

Step 2: If A_k satisfies the following two conditions (1) linearly independent of any other vectors in C' and (2) either $A_kD>\bar{0}$ or $-A_kD>\bar{0}$, then either $\Pi_l=\alpha A_k$ (if $A_kD>\bar{0}$) or $\Pi_l=-\alpha A_k$ (if $-A_kD>\bar{0}$), $C'=C'\cup\{\Pi_l\}$, where α is such that the greatest common divisor of entries of Π_l is unity and Π_l is integral, and $l=l+1$.

Step 3: $k=k+1$. If all rows of A have not been considered, go to Step 2.

Step 4: Pick an unprocessed combination of two linearly independent dependence vectors $\bar{d}_i$ and $\bar{d}_k$ form D. Let Π^* be the integral solution of equation $\Pi(\bar{d}_i-\bar{d}_k)=0$ such that the greatest common divisor of its entries is unity. If Π^* is feasible (either $\Pi^*D>\bar{0}$ or $-\Pi^*D>\bar{0}$), then $\Pi_l=\Pi^*$, $C'=C'\cup\{\Pi_l\}$, and $l=l+1$

Step 5: If all distinct combinations of two linear independent dependence vectors are processed, stop. Otherwise, go to Step 4.

The procedure of finding the optimal solution Π^o for n-dimensional algorithms and the formal proof of the correctness of these procedures can be found in [13]. The execution of the generalized selective shrinking can be coded using parallel constructs which is discussed next.

3.3. Code Conversion

After the optimal schedule vector Π^o is found, the parallel version of the code, using DOALL or DOACROSS [3] constructs, can be generated from the original serial code as explained next.

Let the original serial program be as in (2.1), the optimal schedule vector $\Pi^o=[\pi_1, \pi_2, ..., \pi_n]$ and $disp\,\Pi^o$ be as defined as in Definition 3.1. Then the parallel code using DOALL construct is as follows:

```
l=min{Π°j̄: j̄ ∈ J}
u=max{Π°j̄: j̄ ∈ J}
DO k=l, u, disp Π°
 DOALL {j̄: Π°j̄=k, k+1, ..., k+disp Π°−1}
   S1(j̄)
   S2(j̄)                                   (3.6)
   ...
   Sp(j̄)
 END
END
```

The statement DOALL $\{\bar{j}: \Pi^o\bar{j}=k, k+1, ..., k+disp\,\Pi^o-1\}$ corresponds to the parallel execution of all iterations $\bar{j}$ such that $\Pi^o\bar{j}=k$, $k+1$, ..., or $k+disp\,\Pi^o-1$. For the algorithm in Figure 1, if $u_2<2u_1$, the optimal schedule vector is $\Pi^o=[0,1]$ and $disp\,\Pi^o=\min\{\Pi^o\bar{d}_i: \bar{d}_i \in D\}=\Pi^o\bar{d}_2=4$. The serial code in Figure 1 can be transformed into the following parallel code:

```
l=min{Π°j̄: j̄ ∈ J}=[0,1][3,5]^T=5
u=max{Π°j̄: j̄ ∈ J}=[0,1][3,u2]^T=u2
DO k=5, u2, 4
 DOALL {j̄: Π°j̄=k, k+1, k+2, k+3}
   S1: A(j1,j2)=B(j1−3,j2−5)
   S2: B(j1,j2)=A(j1−2,j2−4)
 END
END
```

The DOACROSS constructs can also be used to express the parallel code of the algorithm in (2.1) with optimal schedule vector $\Pi^o=[\pi_1, \pi_2, ..., \pi_n]$.

```
DO j1=l1, u1
 delay ((j1−1) * π1)
 DO j2=l2, u2
  delay ((j2−1) * π2)
  ...
   DO jn=ln, un
    delay ((jn−1) * πn)
      S1(j̄)                                (3.7)
      S2(j̄)
      ...
      Sp(j̄)
   END
  ...
 END
END
```

According to the definition of DOACROSS [3], the computation in iteration $\bar{j}=[j_1, ..., j_n]^T$ is executed at time step $\Pi\bar{j}$.

3.4. Relation to Wavefront Scheduling

The wavefront method [16] (or hyperplane method [6]) is used to execute nested loops on parallel and vector machines when no loop can be parallelized independently of the others in a loop nest. In this method, a set of wavefronts or hyperplanes is drawn across the index set and all iterations on the same wavefront (hyperplane) can be executed in parallel. The only difference between the wavefront method and a feasible generalized selective shrinking with schedule vector Π is that the latter executes several (*disp* Π) wavefronts simultaneously, exploiting more parallelism in the algorithm. In addition, the method presented in this paper provides a technique to find an *optimal* wavefront schedule.

4. True Dependence Shrinking

This section informally describes *true dependence shrinking* and motivates the generalized definition. The relationship between generalized selective shrinking and *generalized true dependence shrinking* is also discussed. A procedure is proposed to find the optimal loop transformation for generalized true dependence shrinking for 2-dimensional algorithms with constant-bounded index sets.

4.1. True dependence and generalized true dependence shrinking

In [11], for an n-dimensional algorithm (J, D), $n>1$, true dependence shrinking transforms the n-dimensional index set into a 1-dimensional index set by coalescing [8], [12] the original nested loops according to their sequential order. Let u_k and l_k be the upper and lower bounds of the *kth* nested loop, then each dependence vector $\bar{d}=[d_1, ..., d_n]^T$ is transformed into the *true dependence distance* [11], the corresponding dependence vector in the new 1-dimensional index space, $\sum_{i=1}^{n} d_i \prod_{q=i+1}^{n} (u_q-l_q+1)$. Let δ be the minimum of all true dependence distances of the algorithm; then the reduction factor[4] is δ, and δ consecutive index points in the new 1-dimensional index set can be executed in parallel without violating dependence relations of the algorithm.

As an example, for the algorithm in Figure 1, according to the true dependence shrinking described in [11], the 2-dimensional index set in Figure 3a is mapped into a 1-dimensional index set as shown in Figure 3b. Clearly, the index points in Figure 3b are ordered from left to right according to the exact sequential execution order of the nested loops in Figure 1. The true dependence distances of $\bar{d}_1$ and $\bar{d}_2$ are $3(u_2-4)+5$ and $2(u_2-4)+4$, respectively. Therefore, the reduction factor is $\delta=2(u_2-4)+4$ and $2(u_2-4)+4$ consecutive index points in the 1-dimensional index set in Figure 3b can be executed in parallel.

True dependence shrinking can be described by the mapping $\tau: J \rightarrow \tau(J)$, $\tau(\bar{j}) = T(\bar{j}-\bar{v})$, $\forall \bar{j} \in J$ where

$$T=[\prod_{i=2}^{n}(u_i-l_i+1), ..., \prod_{i=k+1}^{n}(u_i-l_i+1), ..., 1] \qquad (4.1)$$

with $\prod_{i=k+1}^{n}(u_i-l_i+1)$ being the *kth* entry of T and $\bar{v}=[l_1, ..., l_n]^T$ is the *shift vector*. An index point $\bar{j} \in J$ is mapped into the new 1-dimensional space as point $T(\bar{j}-\bar{v})$ and the dependence matrix D is mapped as $TD = [T\bar{d}_1, ..., T\bar{d}_m]$. Therefore, the transformed algorithm can be considered a 1-dimensional algorithm with index set $\tau(J)$ and dependence matrix TD. Clearly, no two or more index points $\bar{j}_1, \bar{j}_2 \in J$ are mapped into the same point in the 1-dimensional index set since $T\bar{j}_1 \neq T\bar{j}_2$, $\forall \bar{j}_1, \bar{j}_2 \in J$. The reduction factor is $\delta = \min\{T\bar{d}_i: i=1, ..., m\}$. Usually, $TD>0$ is required otherwise the new 1-dimensional algorithm is not computable. In the new 1-dimensional algorithm, δ consecutive points can be executed in parallel. For the algorithm in Figure 1, the mapping matrix is $T=[u_2-4, 1]$ and the shift vector is $\bar{v}=[3, 5]^T$. An index point $\bar{j} \in J$ is mapped into the 1-dimensional index set as point $T(\bar{j}-\bar{v})$. The new dependence matrix is $[u_2-4, 1]D = [3u_2-7, 2u_2-4]$ and the reduction factor is $\delta=\min\{3u_2-7, 2u_2-4\}$.

Again, some interesting observations can be made from the example above. Consider the algorithm in Figure 5 with a dependence vector $\bar{d}=[0, 1]^T$. By true dependence shrinking in [11], the corresponding true dependence distance is $T\bar{d}=1$ where $T = [u+1, 1]$ is as defined in Equation 4.1 and the reduction factor $\delta=\min\{T\bar{d}_i, i=1, ..., m\} = 1$. Thus, the new algorithm $(\tau(J), TD)$ can only be executed sequentially and no speedup is possible. However, it is clear that if another mapping τ specified by a matrix T, different from the one defined in Equation 4.1, it is possible to obtain speedup and explore the maximal degree of parallelism.

The total execution time by the true dependence shrinking specified by T is

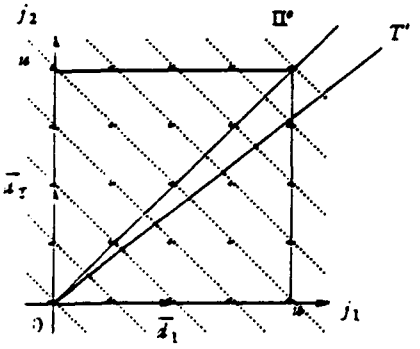
$$t = \left\lfloor \frac{\max\{T(\bar{j}_1-\bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\}+1}{\min\{T\bar{d}_i: i=1, ..., m\}} \right\rfloor = \left\lfloor \frac{\max\{T(\bar{j}_1-\bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\}}{\min\{T\bar{d}_i: i=1, ..., m\}} \right\rfloor + 1 \qquad (4.2)$$

Instead of the mapping matrix T in Equation 4.1, T should be chosen such that the total execution time t is minimized. This is illustrated by the example in Figure 6a where

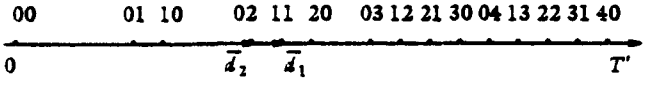
$$J=\{\bar{j}: 0 \leq j_1, j_2 \leq u, \bar{j} \in Z^2\} \text{ and } D=[\bar{d}_1, \bar{d}_2]=\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

If the mapping matrix $T = [1+u, 1]$ in Equation 4.1 is applied, then the total execution time is $\left\lfloor \frac{[1+u, 1](\frac{u}{u})+1}{2} \right\rfloor = \left\lfloor \frac{u(u+2)+1}{2} \right\rfloor$. Consider another mapping matrix $T' = [1+u, u]$. Clearly $T'D = [2(1+u), 2u] > \bar{0}$ and it can be shown that no two or more index points are mapped into the same image in the new index set (this will become clear later in Section 4.2). Hence, this mapping is feasible. The new 1-dimensional index set is shown in Figure 6b. The total execution time is $t(T') = \left\lfloor \frac{[1+u, u](\frac{u}{u})+1}{2u} \right\rfloor = \left\lfloor \frac{u(2u+1)+1}{2u} \right\rfloor$ which is clearly much shorter than the mapping matrix $[1+u, 1]$ when $u>1$. There may be several choices for mapping matrix T which have shorter execution

[4] The minimum reduction factor δ is dependent on the loop bounds, which may not be known until run-time; a run-time test could be generated to determine the minimum δ.



(a) *The index set before optimal generalized true dependence shrinking.*



(b) New 1-dimensional index set after optimal true dependence shrinking. The point with label *ab* is the image of point $\binom{a}{b}$ in (a).

Figure 6: Illustration of optimal true dependence shrinking.

time than the mapping matrix defined in Equation (4.1). In other words, index points should be allowed to project along not only the *nth* dimension but also other directions. These observations motivate the definition of the *generalized true dependence shrinking* as follows.

Definition 4.1 (generalized true dependence shrinking): *Generalized true dependence shrinking* is a mapping $\tau: J \rightarrow \tau(J)$, $\tau(\bar{j}) = T(\bar{j}-\bar{v})$, $\forall \bar{j} \in J$ where $T=[t_1, t_2, ..., t_n] \in Z^{1\times n}$ is the *mapping matrix*, and $\bar{v}=[l_1, ..., l_n]^T$ is the *shift vector*. Mapping τ must satisfy the following three conditions:

(1) $\forall \bar{j}_1, \bar{j}_2 \in J, \bar{j}_1 \neq \bar{j}_2, T\bar{j}_1 \neq T\bar{j}_2$,
(2) $TD > 0$, and
(3) $gcd(t_1, t_2, ..., t_n)=1$.

True dependence shrinking in [11] is a special case of generalized true dependence shrinking with T as defined in Equation 4.1; which may not be optimal for some algorithms. Condition 1 in the above definition guarantees that no two or more index points in J are mapped into the same point in the new 1-dimensional index set $\tau(J)$. For any $\bar{j}_1, \bar{j}_2 \in J$, $\bar{j}_1 \neq \bar{j}_2$, if $\tau(\bar{j}_1)=\tau(\bar{j}_2)$ then the mapping τ is said to have *conflicts*. Otherwise, mapping τ is *conflict-free*. Condition 2 guarantees that all dependence vectors are transformed into positive true dependence distances and the new algorithm is executable. Because $t(\alpha T) = t(T)$ where $\alpha \neq 0$ is an integer constant, condition 3 avoids repeated representations of solutions. Hence, to find optimal generalized true dependence shrinking is to find an integral row vector T which minimizes the total execution time and satisfies certain constraints:

$$\min \quad f = \frac{\max\{T(\bar{j}_1-\bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\}}{\min\{T\bar{d}_i: i=1, ..., m\}} \qquad (4.3)$$

$$\textit{subject to} \quad \begin{cases} (1)\ \forall \bar{j}_1, \bar{j}_2 \in J, \bar{j}_1 \neq \bar{j}_2, T\bar{j}_1 \neq T\bar{j}_2 \\ (2)\ TD > \bar{0} \\ (3)\ gcd(t_1, ..., t_n)=1 \end{cases}$$

Notice that the total execution time $t=\lfloor f \rfloor+1$. So t is minimized *iff* f is minimized.

4.2. Conflict-free mappings

The concept of conflict-free mapping is well defined and explained in [15]. Necessary and sufficient conditions for mapping matrix $T \in Z^{k\times n}$, $k<n$ to have $T\bar{j}_1 \neq T\bar{j}_2$, $\forall \bar{j}_1, \bar{j}_2 \in J$, $\bar{j}_1 \neq \bar{j}_2$ are also given in [15]. For clearity, it is necessary to review one definition and one theorem in [15].

Let $T \in Z^{1\times n}$ be the mapping matrix from an n-dimensional index set J to a 1-dimensional index set $\tau(J)$. A conflict occurs if two or more index points are mapped into the same point in $\tau(J)$. That is, for two distinct index points $\bar{j}_1, \bar{j}_2 \in J$, if $T\bar{j}_1 = T\bar{j}_2$, then there is a conflict. Consider a non-zero vector $\bar{\gamma}$ such that $T\bar{\gamma}=0$. Let $\bar{j}_1=\bar{j}_2+\bar{\gamma}$, then $T\bar{j}_1 = T\bar{j}_2$. If both $\bar{j}_1$ and $\bar{j}_2$ belong to the index set, then $\bar{j}_1$ and $\bar{j}_2$, are mapped to the same point in the 1-dimensional index set $\tau(J)$ and a conflict occurs. One possible way to avoid conflicts is to find the mapping matrix T such that, for any arbitrary index point $\bar{j} \in J$ and any γ that is a non-zero integral solution of equation $T\gamma=0$, $\bar{j}+\gamma$ does not belong to the index set J. This concept is illustrated by Figure 7 which shows a 2-dimensional index set $J=\{[j_1, j_2]^T: 0 \leq j_1, j_2 \leq 4, j_1, j_2 \in Z\}$. If $\bar{\gamma}$ is $\bar{\gamma}_1=[1,1]^T$, then index points $\bar{j}=\bar{0}$ and $\bar{j}+\bar{\gamma}_1=[1,1]^T$ both belong to index set J and index points $[0,0]^T$, $[1,1]^T$, $[2,2]^T$, ..., $[4,4]^T$ will be mapped into the same point in $\tau(J)$. Therefore, there is at least one conflict. However, if γ is $\bar{\gamma}_2=[3,5]^T$, there will be no conflict at all because for any arbitrary $\bar{j} \in J$, $\bar{j}+\bar{\gamma}_2 \notin J$. Intuitively, if vector $[3,5]^T$ is drawn with one end at $[0,0]^T$ (or at any other index point of the index set), then the other end is out of the index set and vector $[3,5]^T$ does not meet any integer points in the index set. Therefore, the mapping with this γ is conflict-free. To describe these concepts formally, the following definitions are introduced.

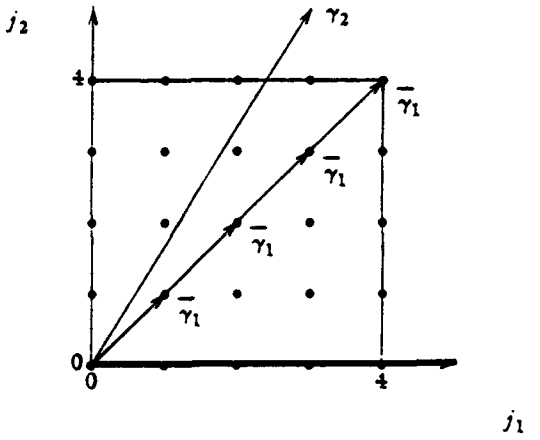


Figure 7: Non-feasible conflict vector $\bar{\gamma}_1$ and feasible conflict vector $\bar{\gamma}_2$. Vector $\bar{\gamma}_2$ does not meet any integral points inside the index set.

Definition 4.2 (Conflict vector, feasible and non-feasible conflict vectors and conflict-free mapping

matrix): Given an algorithm (J, D) and a mapping matrix $T \in Z^{1 \times n}$, an integral column vector $\gamma = [\gamma_1, ..., \gamma_n]^T$ is a *conflict vector* of the mapping matrix T if and only if $T\gamma = 0$ and $gcd(\gamma_1, ..., \gamma_n) = 1$. If for any arbitrary index point $\bar{j} \in J$, $\bar{j} + \gamma \notin J$, then γ is a *feasible conflict vector*. If there exists at least one index point $\bar{j} \in J$ such that $\bar{j} + \gamma \in J$ then γ is called a *non-feasible conflict vector*. If all the conflict vectors are feasible, then T is *conflict-free*.

For algorithms with constant-bounded index sets, the common characteristics of feasible conflict vectors are described in the following theorem.

Theorem 4.1: For algorithms with constant-bounded index sets defined by Equation 3.4, a mapping matrix T is conflict-free if and only if for each of its conflict vectors $\gamma = [\gamma_1, ..., \gamma_i, ..., \gamma_n]^T$ there exists an entry γ_i such that $|\gamma_i| > u_i - l_i$.

Proof: (=>). Without loss of generality, let $l_i = 0$, $i = 1, ..., n$. Because T is conflict-free, all the conflict vectors of T are feasible. Now suppose that γ is a conflict vector of T and $|\gamma_i| \leq u_i$, $i = 1, ..., n$. Consider the index point $\bar{j} = [j_1, ..., j_n]^T$ where $j_i = 0$ if $\gamma_i \geq 0$ and $j_i = -\gamma_i$ if $\gamma_i < 0$. It is clear that both $\bar{j}$ and $\bar{j} + \gamma$ belong to the index set J defined by Equation 3.4 because $|\gamma_i| \leq u_i$, $i = 1, ..., n$. By Definition 4.2, γ is not feasible which is contrary to the assumption. Therefore, for each of the conflict vectors γ, there must exist an entry γ_i such that $|\gamma_i| \geq u_i$.
(<=). Let γ be a conflict vector of mapping matrix T and consider an arbitrary index point $\bar{j}$ belonging to the index set defined by Equation 3.4. Let $\bar{j}' = \bar{j} + \gamma = [j'_1, ..., j'_n]^T$. Because there exists an entry γ_i of γ such that $|\gamma_i| > u_i$ and $u_i \geq j_i \geq 0$, $j'_i = j_i + \gamma_i > u_i$ if $\gamma_i > 0$ and $j'_i = j_i + \gamma_i < 0$ if $\gamma_i < 0$. In both cases, $\bar{j}'$ is not in the index set J and γ is feasible. This implies that T is conflict-free. □

Example 4.1: Consider a 3-dimensional index set $J = \{\bar{j}: 0 \leq j_i \leq u, i = 1, 2, 3\ \bar{j} \in Z^3\}$, mapping matrix $T = [u+1, u, 1]$ and the following solutions of $T\gamma = 0$: $\gamma_1 = [-u, u+1, 0]^T$ and $\gamma_2 = [1, -1, -1]^T$. Clearly, $T\gamma_1 = T\gamma_2 = 0$ and the greatest common divisors of their entries are unity. So γ_1 and γ_2 are conflict vectors of mapping matrix T. However, vector $[2, -2, -2]^T$ is also a solution of equation $T\gamma = 0$ but is not a conflict vector of mapping matrix T because the greatest common divisor of its entries is not unity. Conflict vector γ_1 is feasible because it can be checked that for any arbitrary index point $\bar{j} \in J$, $\bar{j} + \gamma_1 \notin J$. Conflict vector γ_2 is not feasible because for the index point $\bar{j} = [0, 1, 1,]^T \in J$, $\bar{j} + \gamma_2 = [1, 0, 0]^T \in J$. Therefore, T is not conflict-free. □

4.3. Finding optimal generalised shrinking for 2-dimensional algorithms

Notice that if constraint 1 in (4.3) is removed, this optimization problem is exactly the same as optimal generalized selective shrinking problem defined in (3.3). Therefore, our approach to find the optimal solution to (4.3) is as follows. First, the optimal solution $T^* = \Pi^*$ to (3.3) without consideration of conflicts is obtained by Procedure 3.1. Without any doubt, this T^* is not conflict-free. Then, as illustrated in Figure 6, a mapping matrix T' is selected from a small neighborhood of T^* such that T is conflict-free and $t(T') = t(T^*)$. Because T^* is the optimal solution without considering conflicts, $t(T^*)$ is a lower bound on the total execution time of any solutions to (4.3): hence, T' must be optimal. This idea is illustrated by the example in Figure 6a as follows.

Consider the algorithm in Figure 6a again, by Procedure 3.1, the optimal linear schedule is $\Pi^* = [1, 1]$. Let $T^* = \Pi^*$ and $T' = [1+u, u]$. Clearly, when u is large enough, T' is in a small neighborhood of T^*. All conflict vectors of mapping matrix T' can be expressed as $\gamma = \lambda[-u, 1+u]^T$ where $\lambda = \mp 1$. Clearly, these conflict vectors are feasible by Theorem 4.1. The corresponding $f(T^*)$, $t(T^*)$, $f(T')$ and $t(T')$, defined in (4.3) and (4.2), are as follows, respectively:

$$f(T^*) = \frac{[1, 1](u)}{2} - \frac{2u}{2} = u$$

$$t(T^*) = \left\lceil \frac{[1, 1](u)+1}{2} \right\rceil = \left\lceil \frac{2u+1}{2} \right\rceil = \left\lceil u + \frac{1}{2} \right\rceil = u + 1$$

$$f(T') = \frac{[1+u, u](u)}{2u} = \frac{u(2u+1)}{2u} = u + \frac{1}{2}$$

$$t(T') = \left\lceil \frac{[1+u, u](u)+1}{u} \right\rceil = \left\lceil \frac{u(2u+1)+1}{2u} \right\rceil = \left\lceil u + \frac{u+1}{2u} \right\rceil = u + 1$$

Clearly, $t(T') = t(T^*)$ so, this mapping is optimal. Intuitively, T' is in a very small neighborhood of T^*. By rotating T^* a little bit, all points can be projected along T' such that no two or more points are projected in to the same position on T'. Because T' and T^* are so close, the difference between $t(T')$ and $t(T^*)$ is zero because of the ceiling function, as defined in (4.2).

In the following, finding an optimal generalized true dependence shrinking for 2-dimensional algorithms with constant-bounded index sets is discussed by using the idea mentioned above. How to find optimal solutions for algorithms with any convex polyhedron index sets is under investigation.

Lemma 1: Let $T = [t_1, t_2, ..., t_n]$ and $gcd(t_1, t_2, ..., t_n) = 1$. Then there exist a positive integer w and an index $i \in \{1, ..., n\}$ such that $gcd(t_1 w, ..., t_i w + 1, ..., t_n w) = 1$.

Proof: Because $gcd(t_1, t_2, ..., t_n) = 1$, T has at least one nonzero entry and there exist two elements t_k and t_l such that $gcd(t_k, t_l) = 1$. Without loss of generality, let $k = 1$, $l = 2$ and $t_1 \neq 0$. Let $w = |t_1| w'$ where w' is an arbitrary positive integer. Consider the pair $(t_1 w, t_2 w + 1)$ and it is shown next that $gcd(t_1 w, t_2 w + 1) = 1$. Suppose $gcd(t_1 w, t_2 w + 1) = \beta$. Then

$$t_1 w = \beta a_1 \quad or \quad t_1 |t_1| w' = \beta a_1 \tag{4.4}$$

and

$$t_2 w + 1 = \beta a_2 \quad or \quad -t_2 w + \beta a_2 = 1 \tag{4.5}$$

According to [7], equation $ax + by = 1$ has integral solution iff $gcd(a, b) = 1$ and $gcd(x, y) = 1$. Therefore, because of Equation 4.5, $gcd(w, \beta) = 1$. By Equation 4.4, β consists of factors of either t_1 or w' or both. In both cases, β is a factor of w. Therefore, it must have $\beta = 1$ otherwise, $gcd(w, \beta) > 1$. Hence, $gcd(t_1 w, t_2 w + 1) = \beta = 1$ which implies $gcd(t_1 w, t_2 w + 1, t_3 w, ..., t_n w) = 1$. □

Lemma 2: Consider a 2-dimensional algorithm with $J = \{\bar{j}: 0 \leq j_i \leq u_i, u_i, j_i \in Z, i = 1, 2\}$. Let $T = [t_1, t_2]$ and $gcd(t_1, t_2) = 1$. Without loss of generality, let $t_1 \neq 0$. Let w be a positive

integer such that (1) $w = |t_1|w'$ for a positive integer w' and (2) $w|t_1| > u_2$. Then mapping matrix $T' = [t_1 w,\ t_2 w + 1]$ is conflict free.

Proof: Let $\bar{\gamma} = [\gamma_1, \gamma_2]^T$ and consider the following equation

$$T'\bar{\gamma} = \bar{0}. \tag{4.6}$$

Clearly, all conflict vectors of T' can be expressed as follows:

$$\begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} = \lambda \begin{bmatrix} t_2 w + 1 \\ -t_1 w \end{bmatrix} \tag{4.7}$$

where λ is a constant such that $\bar{\gamma}$ is integral and $gcd(\gamma_1, \ldots, \gamma_n) = 1$. By Lemma 1, $gcd(t_2 w + 1, -t_1 w) = 1$. Hence, λ must be either -1 or 1 to have the above two conditions satisfied. Because, $|\gamma_2| = |t_1|w > u_2$, for any value of λ, $\bar{\gamma}$ is a feasible conflict vector by Theorem 4.1. So T' is conflict-free. □

Theorem 4.2: Consider a 2-dimensional algorithm (J, D) with index set $J = \{\bar{j}: 0 \leq j_i \leq u_i,\ u_i, j_i \in Z,\ i = 1, 2\}$. Let $T^* = [t_1, t_2]$ be the optimal solution to (3.3) found by Procedure 3.1. Without loss of generality, let $t_1 \neq 0$. Then $T' = [t_1 w,\ t_2 w + 1]$ is the optimal solution to problem (4.3) where w is a positive integer such that (1) w is a multiple of $|t_1|$, (2) $w > u_2/|t_1|$ and (3) w is reasonably large.

Proof: By Lemmas 1 and 2, T' is conflict-free and the greatest common divisor of the entries of T' is unity. Let $\bar{d}_r$ be such that $\min\{T^* \bar{d}_i: i = 1, \ldots, m\} = T^* \bar{d}_r$ and $\bar{p}_h$ be such that $\max\{T^*(\bar{j}_1 - \bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\} = T^* \bar{p}_h$. Then

$$f(T^*) = \frac{\max\{T^*(\bar{j}_1 - \bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\}}{\min\{T^* \bar{d}_i: i = 1, \ldots, m\}} = \frac{T^* \bar{p}_h}{T^* \bar{d}_r}.$$

Because T' is in a very small neighborhood of T^* and w is reasonably large, $\min\{T' \bar{d}_i: i = 1, \ldots, m\} = T' \bar{d}_r$ and $\max\{T'(\bar{j}_1 - \bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\} = T' \bar{p}_h$ when $w \to \infty$. Hence

$$f(T') = \frac{\max\{T'(\bar{j}_1 - \bar{j}_2): \bar{j}_1, \bar{j}_2 \in J\}}{\min\{T' \bar{d}_i: i = 1, \ldots, m\}} = \frac{T' \bar{p}_h}{T' \bar{d}_r}.$$

$T' \bar{d}_r = (wT^* + [0, 1])\bar{d}_r = wT^* \bar{d}_r + d_{2r}$ and $T' \bar{p}_h = (wT^* + [0, 1])\bar{p}_h = wT^* \bar{p}_h + p_{2h}$. So

$$f(T') = \frac{wT^* \bar{p}_h + p_{2h}}{wT^* \bar{d}_r + d_{2r}}$$

and

$$f(T') = f(T^*) + \frac{p_{2h} T^* \bar{d}_r - d_{2r} T^* \bar{p}_h}{wT^* \bar{d}_r (wT^* \bar{d}_r + d_{2r}) T^* \bar{d}_r}.$$

Therefore, when w is large enough, $\lfloor f(T') \rfloor = \lfloor f(T^*) \rfloor$ and $t(T') = t(T^*)$. □

Notice that from (3.3) and (4.3), the total execution time by an optimal generalized true dependence shrinking cannot be shorter than optimal generalized selective shrinking. In the selective shrinking, the original index set can be thought as being transformed into a 1-dimensional index set by the mapping Π^*. A single point in the new 1-dimensional index set is the image of multiple points in the original index set which can be executed in parallel without violating dependence relations. Optimal true dependence shrinking is basically the same mapping as the selective shrinking except that it requires a one-to-one mapping. In some cases, the requirement of one-to-one mapping is not necessary if the purpose is to identify all computations which can be executed in parallel.

5. Conclusions

In this paper, two loop transformation techniques proposed in [11] are generalized in order to explore more parallelism of algorithms with nested loops. Methods to find optimal transformations are discussed. For the generalized selective shrinking, the method finds time-optimal transformations for algorithms with any convex polyhedron index sets. For the generalized true dependence shrinking, the method finds optimal solutions for 2-dimensional algorithms with constant-bounded index sets. How the two transformation techniques are related is also briefly discussed.

References:

[1] M.S. Bazaraa and C.M. Shetty, *Nonlinear Programming: Theory and Algorithms*, John Wiley & Sons 1979.

[2] M. Byler, J. R.B. Davies, C. Huson, B. Leasure, M. Wolfe, "Multiple Version Loops," *Proc. Int. Conf. on Parallel Processing*, pp. 312-318, August 1987, St. Charles, IL.

[3] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," *Proc. Int. Conf. on Parallel Processing*, pp. 836-844, August 1986, St. Charles, IL.

[4] J.A.B. Fortes and B.W. Wah, "Systolic Array — From Concept to Implementation," *IEEE Computer*, July 1987, pp. 12-17.

[5] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM*, Vol. 14, No. 3, pp. 563-590, July 1967.

[6] L. Lamport, "The Parallel Execution of DO Loops," *Comm. of the ACM*, Vol. 17, No. 2, Feb. 1974, pp. 83-93.

[7] L.J. Mordell, *Diophantine Equations*, Academic Press, New York, 1969, pp. 30.

[8] M. T. O'Keefe and H. G. Dietz, "Loop Coalescing and Scheduling for Barrier MIMD Architectures," submitted June 1990 to *IEEE Trans. on Parallel and Distributed Systems*.

[10] J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," *IEEE Trans. on Computers*, Vol. 38, No. 8, pp. 1203-1211, August 1989.

[11] C.D. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and their Impact on Architecture Design," *IEEE Trans. on Computers*, Vol. 37, No. 8, August 1988, pp. 991-1004.

[12] C. D. Polychronopoulos, *Parallel Programming and Compilers*. Boston: Kluwer Academic Publishers, 1988.

[13] W. Shang and J.A.B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *Proceedings of Int. Conf. on Systolic Arrays*, May 1988, pp. 393-402 (also to appear in *IEEE Trans. on Computers*).

[14] W. Shang and J.A.B. Fortes, "On Optimality of Linear Schedules," *Journal of VLSI Signal Processing*, 1, pp. 209-220 (1989), Kluwer Academic Publishers, Boston.

[15] W. Shang and J.A.B. Fortes, "Time-Optimal and Conflict-Free Mappings of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays," *Proc. of Int'l Conf. on Parallel Processing*, August 1990, pp. 101-110 (I) (also to appear in *IEEE Trans. on Parallel and DIstributed Systems)*.

[16] M. Wolfe, "Loop Skewing: The Wavefront Method Revisited," *Int'l J. of Parallel Programming*, Vol. 15, No. 4, 1986, pp. 279-293.

[17] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Cambridge, MA: MIT Press, 1989.

REFERENCE NO. 17

Cam, H. and Fortes, J. A. B., "Fault-Tolerant Self-Routing Permutation Networks," 1992 International Conference on Parallel Processing, August 17-21, St. Charles, Illinois, Vol. I, pp. 243-247.

Note - this paper proposes a novel fault-tolerant interconnection network capable of implementing any permutation using distributed destination-tag based self-routing of messages. The time needed to implement any permutation is the smallest of any network with comparable hardware complexity.

FAULT-TOLERANT SELF-ROUTING PERMUTATION NETWORKS*

Hasan Cam and Jose A.B. Fortes
School of Electrical Engineering
Purdue University, West Lafayette, IN 47907 (USA)

Abstract -- *Four different self-routing permutation networks are presented in this paper. The first self-routing permutation network is constructed from concentrators and digit-controlled* 2×4 *switches. This self-routing permutation network has* $O(\log^2 N)$ *gate-level delay with a very small constant and use* $O(N^2 \log N)$ *VLSI-level hardware.*(*) *The propagation time of this network is less than that of any self-routing permutation network presented to date. The second self-routing permutation network is obtained from the first self-routing permutation network by replacing* 2×4 *switches at some stages by short-circuit switches and removing the concentrators at these stages. This network is proposed to avoid having very large concentrators when* N *gets large. The third self-routing permutation network consists of a* $\log N$*-stage network followed by one concentrator whose outputs are connected to the inputs of the network. Finally, a 1-fault-tolerant self-routing permutation network is obtained by adding intrastage links, multiplexers and demultiplexers to the first self-routing permutation network.*

1. INTRODUCTION

An interconnection network (IN) with $N = 2^n$ inputs/outputs is called a permutation network (or rearrangeable network) if it realises every one of the $N!$ permutations in a single pass. This paper presents four different self-routing permutation networks constructed from concentrator(s) and different digit-controlled switches. Each of these networks realises a given permutation using the destination tag routing scheme. Three of these networks are based on a new approach which does not allow any conflict to occur in switches. Because the sorting is the rearrangement of items into ascending or descending order [2], these networks can also be used as sorting networks.

Sorting networks are often constructed using two-input, two-output comparators that send the smaller of their two inputs to their upper output. A comparator compares its two $\log N$-bit inputs to determine its state. Because a comparator can be replaced by $\log N$ 1-bit wide comparators, the gate-level (also referred as bit-level) delay of sorting networks with N inputs/outputs can be determined by multiplying their comparator-level delay by $\log N$ [1,2]. Although the delay of the self-routing permutation networks presented in this paper are stated in gate-level the area complexities of our self-routing permutation networks are expressed in VLSI-level instead of gate-level because the concentrators used in these networks contain gates whose sizes vary with the number of inputs.

Batcher [3] presented both the bitonic network and the odd-even network, both of which take $O(\log^2 N)$ comparator-level time to sort input vectors of size N. Ajtai et al. [4] introduced a network of size $O(N \log N)$ that can sort in $O(\log N)$ steps, which is asymptotically superior to the existing sorting networks, but not practical because of its large constant. Koppelman and Oruc [1] described a self-routing permutation network which has $O(\log^3 N)$ gate-level delay and uses $O(N \log^3 N)$ gate-level hardware. Jan and Oruc [9] presented two self-routing permutation networks, one of which has $O(N \log^2 N)$ gate-level cost and $O(\log^3 N)$ gate-level delay. If VLSI-area (instead of simply gate-level area) is considered, then Jan and Oruc's network has $O(N^2 \log N)$ VLSI-level cost as a consequence of the results in [10,11] which show that cube-type networks use $O(N^2)$ layout-area. One of the self-routing permutation networks presented in this paper has $O(\log^2 N)$ gate-level delay and $O(N^2 \log N)$ VLSI-level cost.

This paper is organised as follows. Section 2 briefly explains radix sorting and the destination routing scheme and points out the close relation between them. In Section 3, the configuration and performance analysis of a self-routing permutation network are described. Section 4 presents three additional self-routing permutation networks which are obtained by modifying the self-routing permutation network presented in Section 3. Section 5 is dedicated to the conclusions.

2. RADIX SORTING AND CONFLICTS

Some self-routing INs, called cube-type networks, employ the so-called destination tag routing scheme for realising any passable permutation through them [7]. In this scheme, the destination address of the ith input, $0 \le i \le N-1$, is used as the routing tag for the ith input and a 2×2 switch at stage k for $1 \le k \le n$ examines the kth bit of the destination address of the incoming input: if the kth bit is 0, then the upper output of the switch is taken; otherwise, the lower output is taken. In some cube-type networks such as the baseline, the way the destination tag routing scheme works is closely related to radix sorting. To see this, let S be the set of the binary representations of N numbers from 0 to $N-1$ in some order. Radix sorting sorts the elements of S

* Research supported in part by the Office of Naval Research under contract No. 00014-90-J-1483 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under contract No. 00014-88-k-0723.

(*) All logarithms are in base 2 unless stated otherwise.

recursively as follows: in the first iteration, the set S is partitioned into two subsets, say S_1 and S_2, such that the most significant bits of all the numbers of S_j, $j \in \{1,2\}$, are the same. In the second iteration, the set S_j is partitioned into two subsets such that the second most significant bits of all the numbers of each subset are the same. The partitioning process of a set into two subsets of equal size is iterated for the third most significant bit, the fourth most significant bit, and finally the least significant bit. Now, let us consider the realisation of a passable permutation through the baseline network using the destination tag routing scheme. The first stage partitions the given set of N numbers into two subsets such that the most significant bits of the destination addresses of all the inputs at the upper (respectively, lower) half of the second stage are 0 (respectively, 1). Similar to the iterations of the radix sorting, the kth stage of the baseline network partitions every incoming subsets of the inputs into two subsets with respect to the kth most significant bits of the destination addresses of their inputs.

Cube-type networks pass a small fraction of $N!$ permutations because, for some permutations, "conflicts" occur in the switches when the kth bits of the destination addresses of the 2 inputs constitute either the set {0,0} or the set {1,1}. The networks proposed in this paper use 2×4 switches to prevent conflicts and are based on the idea of radix sorting. This implementation is different from that of Jan and Oruc [9] in many respects, although both implementations have some common points. The network proposed by Jan and Oruc first demultiplexes each input onto all N outputs of a distributor through a binary tree of $N-1$ vertices, then the second half of the network concentrate these inputs to their appropriate outputs. The approach of this paper replaces the 2×2 switches by the 2×4 switches to enable the 2 inputs of every switch to go to next stage simultaneously even if their control bits constitute the set {0,0} or {1,1}. But, no matter what the 2 inputs of a 2×4 switch are, the 2 outputs of any switch carry invalid messages to the next stage. In order to detect and discard these invalid messages before they arrive at the switches of next stage, concentrators are provided between stages.

3. SELF-ROUTING PERMUTATION NETWORK

If a network sets its swicthes using a distributed routing scheme in which each switch determines its own setting dynamically by examining the routing tag bits of the inputs, then this network is said to be self-routing. In this section, a self-routing permutation network constructed from concentrators and digit-controlled 2×4 switches is presented. First, the definitions of concentrator and hyperconcentrator are given below.

An *(N,M) concentrator* is an IN that has N inputs $X_1, X_2, \ldots, X_N$ and $M < N$ outputs $Y_1, Y_2, \ldots, Y_M$, and can establish M disjoint paths from any set of M inputs to the M outputs [6]. An IN with N inputs and N outputs is called a *N-hyperconcentrator* if there are R disjoint paths, for any $1 \le R \le N$, from each set of R inputs to the first R outputs $Y_1, Y_2, .., Y_R$. This implies that any N-hyperconcentrator can be used as a (N,M) concentrator by simply choosing the first M outputs of the hyperconcentrator as the M outputs of the concentrator. In this paper, the hyperconcentrator of Cormen and Leiserson [6] is used as a concentrator.

3.1. Network Configuration

The first self-routing permutation network introduced in this paper is called PN and can be constructed recursively. The first iteration of the recursive construction, shown in Figure 1, is composed of the input and output stages. The input stage consists of $N/2$ digit-controlled 2×4 switches and two $(N,N/2)$-concentrators. The output stage consists of two copies of a $N/2$-input PN in parallel. The PNs with smaller inputs/outputs are decomposed recursively until $N/2$ switches of size 2×2 are obtained in the last iteration. (Note that at the last stage of PN, there is no need to employ 2×4 switches because the outputs of a switch are connected to only 2 outputs of PN). This implies that a PN with N inputs/outputs can be constructed recursively in $\log N$ iterations. Thus, PN is composed of $\log N$ stages labeled from left to right starting with 1. For $1 \le k \le n-1$, the stage k consists of 2^{n-1} 2×4 switches followed by 2^k $(2^{n+1-k}, 2^{n-k})$ concentrators numbered from 0 to 2^k-1. The stage n consists of 2^{n-1} 2×2 switches. As an example, Figure 2 illustrates PN with N=8 inputs/outputs.

3.2. Routing Scheme

All the inputs routed by PN to their destinations are bit-serial data streams in a packet format. Each packet is divided into header and data sections; the header contains the destination address of the packet and a single "valid bit", which is "1" if the packet contains valid message and "0" if it contains invalid message. Those 2 outputs of any 2×4 switch which are not connected to any of its inputs contain invalid messages. To inform the concentrator(s) that they carry invalid messages, the switch appends a valid bit of 0 to the header of each invalid message. On the other hand, the valid bits of the other 2 outputs that are connected to the incoming inputs are made 1. These valid bits of the inputs are used during setup the conducting paths of a concentrator, so that a valid message arrives at an output of the concentrator as opposed to an invalid message which is not routed any output of the concentrator. Also, the concentrators used in PN require all n bits of each invalid message to be 0's.

In PN, the destination tag routing scheme is employed to realize any permutation through it. The switches and the concentrators of PN are "self-setting" when inputs are presented to them. No matter what the inputs are, the switches never have a conflict because both inputs can be routed to the same concentrator. For $1 \le k \le n-1$, if the kth most significant bit of the destination tag of the upper input to a switch at the kth stage is 0, then the upper input of the switch is routed to the first (topmost) output; otherwise, the upper input is routed to the third output. If the kth most significant bit of the destination tag of the lower input to a switch at the kth stage is 0, then the

lower input of the switch is routed to the second output from top; otherwise, the lower input is routed to the fourth output. If both kth bits of the inputs are 0's, then both inputs are routed to the upper two outputs, called upper-connection; if both kth bits are 1's, then they are directed to the lower two outputs, called lower-connection. The input links of a 2×4 switch are labeled by 0 and 1, while the output links are numbered 0, 0, 1, and 1 from top to bottom (see Figure 3 for the illustration of the four states of a 2×4 switch).

3.3. Concentrators Used in the PN

The PN uses the M-hyperconcentrator, $M = 2^m$ and $2 \leq m \leq n$, of Cormen and Leiserson [6] as an $(M,M/2)$ concentrator by simply choosing the first $M/2$ outputs of the M-hyperconcentrator as the $M/2$ outputs of the $(M,M/2)$ concentrator. Because an $(M,M/2)$ concentrator in PN have exactly $M/2$ valid and $M/2$ invalid messages at its inputs, every output of the concentrator carries a valid message to the next stage. Their M-hyperconcentrator routes all the valid messages to the first outputs and does not route any of the invalid messages at all.

Theorem 1. The network shown in Figure 1 is a self-routing permutation network.

Proof. Each of the 2×4 switches at the first stage of Figure 1 examines the most significant bits of the destination addresses of its inputs to set itself to one of the four states shown in Figure 3. Because a 2×4 switch is connected to each of the two concentrators through 2 links, no conflict occurs in the switches. So, a 2×4 switch routes the input whose destination address' most significant bit equals 0 (respectively, 1) to the upper concentrator (respectively, the lower concentrator).

Because the realization of permutations on N integers through the network is considered, the most significant bits of the $N/2$ destination addresses are 1 and the most significant bits of the other half is 0. Therefore, each of the two $(N,N/2)$ concentrators has $N/2$ valid and $N/2$ invalid messages. These concentrators route their only valid messages to their outputs. Finally, each of the 2 permutation networks at the last stage of Figure 1 route their inputs to their destinations. Therefore, the network shown in Figure 1 is rearrangeable and, hence, is a permutation network. Due to the fact that the concentrators are self-routing and the switches set themselves "online" by examining the destination addresses of the inputs, the network illustrated in Figure 1 is also self-routing. □

3.4. Performance Analysis

In this section, the hardware cost and the propagation delay of the PN are computed. In deriving the computations, the propagation delay of any switch used in PN will be taken as 1. In other words, the propagation delay of a digit-controlled 2×4 switch is the basic unit of time. Let A_{PN} and D_{PN} denote the hardware cost and the propagation delay of PN, respectively.

3.4.1. The Propagation Delay, D_{PN}

Because PN can be constructed recursively, its propagation delay can be described with the help of Figure 1 by the following recurrence equation:

$$D_{PN}(N) = D_{PN}(N/2) + D_C(N,N/2) - D_S \quad (3.1)$$

where $D_{PN}(N)$ is the propagation delay of PN with N inputs/outputs, $D_C(N,N/2)$ is the propagation delay of an $(N,N/2)$ concentrator, and D_S is the delay of a 2×4 switch which equals 1. Because PN uses the N-hyperconcentrator of Cormen and Leiserson [6] as a $(N,N/2)$ concentrator, $D_C(N,N/2)$ has $2\log N$ gate-level delay [6], i.e. $D_C(N,N/2) = 2\log N$. So, the equation (3.1) can be rewritten

$$D_{PN}(N) = D_{PN}(N/2) + 2\log N + 1 \quad (3.2)$$

$$= \log^2 N + 2\log N - 2 \quad (3.3)$$

Hence, realising a permutation through PN takes $O(\log^2 N)$ gate-level delay.

3.4.2. The Hardware Cost, A_{PN}

The cost is given by the recurrence:

$$A_{PN}(N) = 2A_{PN}(N/2) + 2A_C(N,N/2) + A_{st} \quad (3.4)$$

where $A_{PN}(N)$ is the cost of PN with N inputs/outputs, $A_C(N,N/2)$ is the cost of an $(N,N/2)$ concentrator, and A_{st} is the cost of $N/2$ switches and all the interconnection links in a stage of the PN. Because the N-hyperconcentrator is used as an $(N,N/2)$ concentrator, the cost of the N-hyperconcentrator $A_H(N,N)$ equals $A_C(N,N/2)$. Because the N-hyperconcentrator of Cormen and Leiserson uses $\Theta(N^2)$ components and has area $\Theta(N^2)$, $A_C(N,N/2) = N^2$.

According to [11], A_{st} for a shuffle-exchange stage is $O(N^2/\log^{3/2} N)$ and $O(N^2/\log N)$, respectively. Therefore, it is assumed that $A_{st} = O(N^2)$. When these values are substituted in (3.4), we obtain

$$A_{PN}(N) = 2A_{PN}(N/2) + 2N^2 + N^2 \quad (3.5)$$

$$= N/2 + 3\,N^2\,(n-1) - 3/2\,N^2 + 3N \quad (3.6)$$

The equation (3.6) shows that PN uses $0(N^2\log N)$ VLSI-level hardware. Table 1 compares the propagation delay and cost of PN with those of the networks of Batcher [3] and Jan and Oruc [9].

4. THREE SELF-ROUTING PERMUTATION NETWORKS

First network: The network PN presented in Section 3 employs the N-hyperconcentrator introduced in [6] at the first stage. This N-hyperconcentrator has $\log N$ stages of merge boxes, each of which routes all its valid messages to the first outputs. The last merge box at the last stage of the N-hyperconcentrator has N inputs and N outputs. Therefore, when N gets larger, implementing the merge boxes at the last stages of the N-hyperconcentrator may require multiple chips. To avoid having large merge boxes, the short-circuit switches introduced in [8] can be used at some of the

first stages of the network. If these switches are used only in a few first stages of PN, the propagation time remains in $O(\log^2 N)$ gate-level delay because the extra delay of these short-circuit switches is a small constant.

Second network: This network, called MPN and shown in Figure 4 for N=16, consists of an n-stage network followed by a concentrator $C(2N,N)$ whose ith output, $0 \le i \le N-1$, is connected back to the ith input. This single $(2N,N)$-concentrator can also be replaced by two $(N,N/2)$ concentrators in parallel. All the switches are digit-controlled, but the switches at stages 2 through n are 4×4, while all the switches at the first stage are 2×4.

Routing Scheme of MPN. The outputs of the concentrator are recycled back $n-1$ times as inputs, so that the given inputs pass through MPN n times. During the kth pass, $1 \le k \le n$, of the inputs through the MPN, the switches are classified into 3 groups as follows: (1) if a switch belongs to a stage whose label is smaller than k, then the switch connects only its upper 2 inputs to its upper 2 outputs and ignores its lower 2 inputs, (2) if a switch belongs to the kth stage, then it determines its state by inspecting the kth bit of the destination tags of its upper 2 inputs in the same way as a switch of PN does (the lower 2 inputs of the switch are ignored), and (3) if a switch belongs to a stage whose label is greater than k, then the switch connects its pth input, $1 \le p \le 4$, to its pth output.

Third network: This network, called APN and illustrated in Figure 5 for N=8, is obtained by adding intrastage links among the switches of PN, multiplexers and demultiplexers at the first stage and the last stage. Each 2×4 switch of PN is replaced by a 3×5 switch shown in Figure 5 which contains 2 additional (auxiliary) links called chain-in link and chain-out link [5]. Chain-in link and chain-out link are the links added to the top and the bottom of a switch, respectively. All $N/2$ switches at stage 1 form a single loop by making use of these auxiliary links. All those switches at stage k, $2 \le k \le n-1$, which receive their inputs from the same concentrator are connected to each other by the auxiliary links to form a loop. The switches at stage n do not need these auxiliary links because the outputs of any concentrator at this stage are connected to only a single switch. After adding intrastage links among the switches of the PN, multiplexers and demultiplexers are also added in the first stage and the last stage, respectively to make the network 1-fault-tolerant. Because of space limitations, the proof that this network is 1-fault-tolerant is not included here and can be found in [12].

APN is a highly robust network in terms of its ability to tolerate multiple switch and link faults simultaneously. When a switch has a fault, the input on the chain-in link bypasses the switch and goes to the next switch in the loop via the chain-out link.

5. CONCLUSIONS

Four different self-routing permutation networks have been presented. One of these self-routing permutation networks, called PN, is constructed from 2×4 switches and concentrators of $2\log N$ gate-level delay. The PN uses the N-hyperconcentrator introduced in [6] as an $(N,N/2)$-concentrator. The PN has $O(\log^2 N)$ gate-level delay and uses $O(N^2 \log N)$ VLSI-level area. The propagation time of this network is less than that of any self-routing permutation network proposed in the literature. In order to make PN 1-fault-tolerant, intrastage links among switches of every stage, along with multiplexers and demultiplexers are added. Using the advantage of recycling the inputs back $n-1$ times, a self-routing permutation network is obtained from an n-stage IN and a concentrator with $2N$ inputs and N outputs. The problem of not being able to implement a concentrator in a single chip when N gets larger has been resolved using short-circuit switches with queues.

REFERENCES

[1] D.M. Koppelman and A.Y. Oruc, "A self-routing permutation network," *J. of Parallel and Distrib. Comput.*, 10, 1990, pp. 140-151.

[2] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1973.

[3] K. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computer Conf.*, vol. 32, 1968, pp. 307-314.

[4] M. Ajtai, J. Komlos, and E. Szemeredi, "An O(nlogn) sorting network," in *Proc. 15th Annu. ACM Symp. Theory Comput.*, 1983, pp. 1-9.

[5] N. Tzeng, P. Yew and C. Zhu, "A fault-tolerant scheme for multistage interconnection networks," *12th Intl. Symp. Computer Architecture*, June 1985, pp. 368-375.

[6] T.H. Cormen and C.E. Leiserson, "A hyperconcentrator switch for routing bit-serial messages," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, 1986, pp. 721-728.

[7] D.H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, no. 12, Dec. 1975, pp. 1145-1155.

[8] Y. Pan and R. Melhem, "Short circuits in buffered multi-stage interconnection networks," *The Computer Journal*, vol. 33, No. 4, 1990, pp. 323-329.

[9] C. Jan and A.Y. Oruc, "Fast self-routing permutation networks," *1991 Int'l Conf. on Parallel Processing*, pp. I-263—I-269.

[10] M.A. Franklin, "VLSI performance comparison of banyan and crossbar communications networks," *IEEE Trans. on Comput.*, vol. C-30, No. 4, Apr. 1981, pp. 283-291.

[11] F.T. Thompson, M. Lepley and G.L. Miller, "Layouts for the shuffle-exchange graph based on the complex plane diagram," *Siam J. Alg.*

Disc. Meth., vol. 5, No. 2, June 1984, pp. 202-215.

[12] Hasan Cam, "Design and permutation routing algorithms of rearrangeable networks," *Ph.D. Thesis*, Purdue Univ., West Lafayette, 1992.

Table 1. Comparison of the performance of 3 self-routing networks, where the delay and cost are computed in gate-level and *VLSI*-level, respectively.

Network	Delay	Cost
Batcher's IN	$O(\log^3 N)$	$O(N^2 \log N)$
Jan & Oruc's IN	$O(\log^3 N)$	$O(N^2 \log N)$
PN	$O(\log^2 N)$	$O(N^2 \log N)$

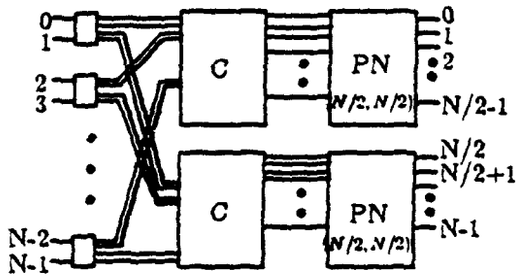


Figure 1. The first iteration of the recursive construction of the self-routing permutation network called *PN*.

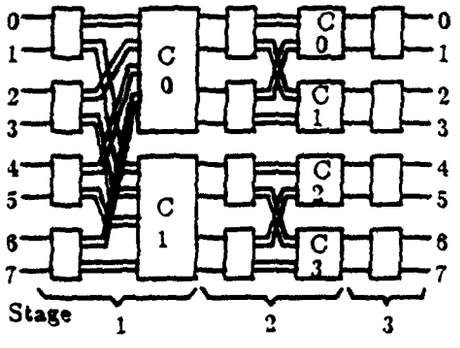


Figure 2. The self-routing permutation network (PN) with $N = 8$. The letter C in the boxes stands for concentrator.

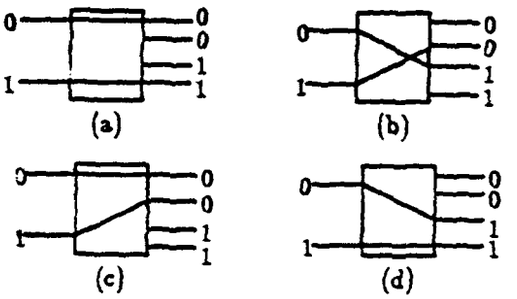


Figure 3. The 4 states of a switching box in *PN*. (a) Straight connection when no conflict exists. (b) Cross connection when no conflict exists. (c) Upper connection when both control bits are 0. (d) Lower connection when both control bits are 1.

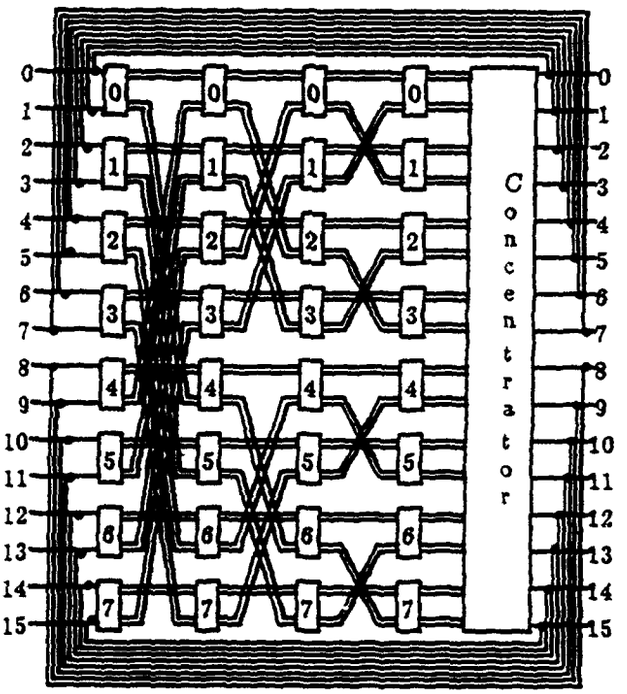


Figure 4. The modified permutation network MPN with N=16 inputs/outputs.

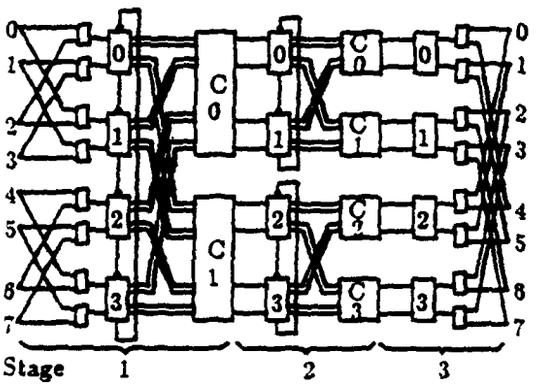


Figure 5. An APN with $N = 8$.

REFERENCE NO. 18

Saghi, G., Siegel, H. J., and Fortes, J. A. B., "On the Viability of a Quantitative Model of System Reconfiguration Due to a Fault," 1992 International Conference on Parallel Processing, August 17-21, St. Charles, Illinois, Vol. I, pp. 233-242.

Note - This paper proposes a quantitative model that can be used by an operating system to decide what reconfiguration strategy should be used. It discusses how the costs of three reconfiguration schemes may be quantifiable in a realistic setting.

ON THE VIABILITY OF A QUANTITATIVE MODEL OF SYSTEM RECONFIGURATION DUE TO A FAULT

Gene Saghi, Howard Jay Siegel, and Jose A. B. Fortes

Parallel Processing Laboratory, School of Electrical Engineering
Purdue University, West Lafayette, IN 47907-1285 USA

Abstract -- *Dynamic fault reconfiguration has been proposed for use in large-scale partitionable parallel processing systems with distributed shared memory. If a processor develops a permanent fault during the execution of a task on a submachine A, three recovery options are migration of the task to another submachine, task migration to a subdivision of A, and redistribution of the task among the fault-free processors in A to effectively "disconnect" the faulty processor. Quantitative models of these three reconfiguration schemes are developed to consider what information is needed to make a choice among these methods for a practical implementation. A multistage cube or hypercube inter-processor network is assumed. It will be pointed out that in certain situations collecting precise values for all needed parameters will be very difficult. Thus, the motivation for this research is to examine to what extent a choice of reconfiguration strategies is quantifiable in a realistic setting.*

1. INTRODUCTION

If massively parallel processing systems are to provide reliable operation over extended periods of time, they must be capable of tolerating faults. A fault-tolerant system must be able to detect and locate faults, to reconfigure itself to "disconnect" and perhaps replace faulty components, to recover from possibly erroneous computations, and to restart operation from a correct state. A dependable fault-tolerant system must be able to both operate in the presence of faults and meet the performance requirements of its applications.

One approach to achieving fault tolerance in parallel processing systems involves the use of redundant hardware. When a faulty component is detected, the system is reconfigured in such a way that the faulty component is replaced by one of the redundant components. An example of such a system is MPP [2]. MPP is a 128 × 128 array of single bit processors implemented as a 64 × 32 array of 2 row × 4 column VLSI ICs. A redundant column of VLSI ICs (64 ICs) is provided to replace any of the 32 columns of ICs within which a faulty processor may exist. Thus, any single processor IC failure in the array can be tolerated. Additional hardware could be added, at additional cost, to allow a greater number of faults to be tolerated. A general disadvantage of the redundant hardware approach is that the extra hardware is idle until a fault occurs.

The work presented here focuses on partitionable parallel processing systems where the set of processors can be partitioned to form multiple independent submachines. The execution of a parallel program on a submachine is defined as a task. It is possible to achieve fault tolerance in such a system by utilizing the reconfigurability of the system to effectively "disconnect" the faulty component. For example, parallel processing systems such as Intel Cube [15], nCUBE [13], IBM RP3 [24], and PASM [29, 8] incorporate partitionable interconnection networks and therefore have the ability to migrate a task from a faulty submachine to a fault-free submachine [e.g., 27]. Such an approach has the advantage of no redundant hardware costs; however, the total available system resources are decreased when a fault occurs. Furthermore, if the smallest possible submachine in such a system consists of more than one processor, the fault-free processors in the faulty submachine become idle. Thus, there is a trade-off between redundant hardware cost and degraded system performance for the fault tolerance schemes discussed.

The architecture assumed here implements a physically distributed memory such that each processor is paired with local memory to form a processing element (PE). Most existing large-scale parallel processing systems use a physically distributed memory approach (e.g., BBN Butterfly [5], Connection Machine CM-2 [32], Intel Cube [15], nCUBE [13], DAP [14], MasPar [4], IBM RP3 [24]). These systems implement either a logically nonshared memory system, a logically shared memory system, or a hybrid of the two memory systems. In a logically nonshared memory system (e.g., CM-2, MasPar), processors cannot access remote memory locations directly. Instead, all communication between PEs is through explicit message passing. In a logically shared memory system (e.g., BBN Butterfly), all system memory appears in the address space of each processor. Accesses to memory locations located in a remote processor's memory requires use of the interconnection network. As a result, remote memory accesses incur a larger latency than local memory references. Careful placement of program code and data is one way to reduce the effects of this network latency. In one type of hybrid memory system (e.g., IBM RP3), a portion of each processor's memory is reserved for nonshared access, the remainder is treated as logically shared memory, and all interproces-

This research was supported by the Office of Naval Research under grant number N00014-90-J-1483 and supported by the Innovative Science and Technology Office of the Strategic Defense Organization and administered through the Office of Naval Research under contract number 00014-88-k-0723.

sor communication is through the shared memory. The work in this paper assumes a logically shared or hybrid memory system.

Previous research on fault recovery by dynamic reconfiguration includes [33] and [34]. These papers explored task redistribution in a MIMD environment to recover from a PE fault for near-neighbor-class problems. Other work has considered the reconfiguration of hypercube architectures in the event of PE failure [12, 20]. Here, quantitative models of three different reconfiguration schemes are developed to consider what information is needed to make a choice among these methods for practical implementations, such as mission critical situations (e.g., automated defense systems). It will be pointed out that, given today's technology, in certain circumstances collecting precise values for all needed parameters will be very difficult. Thus, the motivation for this research is to examine to what extent a choice of reconfiguration strategies is quantifiable in a realistic setting.

The system and fault models used to analyze fault-recovery options are described in Section 2. Section 3 presents recovery options and associated costs. One cost that must be considered for every fault recovery option is the remaining execution time of the task that was on the faulty submachine when the fault occurred. As an example of the difficulties in determining some of these costs, Sections 4 and 5 provide models for determining the remaining execution time of a task depending on the system configuration and operation modes. In Section 4, tasks whose execution times are data independent are considered, while in Section 5 tasks whose execution times are data dependent are considered.

2. SYSTEM MODEL

Two approaches to massively parallel computation are the SIMD and MIMD modes of parallelism [9]. In machines that operate in MIMD mode, the PEs contain programs and data, and each PE functions independently. In machines that operate in SIMD mode, instructions are broadcast to the PEs, the PEs contain only data, and all the PEs operate in lock-step fashion. A multiple-SIMD machine is an SIMD machine that can operate as one or more independent SIMD machines [22]. A mixed-mode machine can operate in either the SIMD or MIMD mode of parallelism and can switch modes at instruction level granularity [8]. A partitionable SIMD/MIMD machine can operate as one or more independent or cooperating submachines, where each submachine may operate as a mixed-mode machine [30].

The analyses here can be used to model MIMD, multiple-SIMD, or partitionable SIMD/MIMD parallel processing systems, utilizing a multistage cube or hypercube interconnection network, and possessing a logically shared or hybrid memory system. The research assumes a partitionable SIMD/MIMD machine with a multistage cube network and can be directly applied to the other cases.

It is assumed that overall system activities are supervised by a dedicated processor known as the system controller unit (SCU), although it could be a program distributed among system processors. Among other duties, the SCU is responsible for allocating and deallocating submachines, and for determining the proper recovery action in the event of a detected fault. The activities in each submachine are supervised by a submachine controller (SC). Similar to the SCU, the SC is assumed to be a designated processor, although it could be a program distributed among the processors of the submachine.

The execution of an SIMD procedure on a submachine is an SIMD process. An MIMD process is the execution of an MIMD procedure on a PE that is part of the submachine. The term subtask refers to a single thread of control (stream of instructions). A subtask can be composed of one or more SIMD processes executed sequentially on the same submachine, or one or more MIMD processes executed sequentially on the same submachine PE. A task can be composed of one or more subtasks. Tasks are coded assuming that any number of *virtual processors* required is available. At execution time, the SC determines the number of physical processors available in the submachine and maps the virtual processors onto physical processors. The ratio of virtual processors to physical processors is called the virtual processor ratio. When the virtual processor ratio is greater than one, some or all of the physical processors will perform the functions of more than one virtual processor. If the virtual processor ratio is less than one, some physical processors will remain idle. Use of a virtual processor scheme allows tasks to be executed on submachines of various sizes without having to be recompiled. The Connection Machine CM-2 [32] uses such a virtual processor scheme to allow programs to be executed on machines consisting of different numbers of physical processors. Other schemes that allow for the same kind of functionality are equally applicable to this research.

The model used for fault tolerance and recovery is as follows. At regular intervals during the execution of a task, the intermediate results of each PE are stored in a different PE within the same submachine (i.e., a permutation of the PEs in the submachine). These results are called checkpoint data and they will be used to restore a valid system state in the event of a fault. Error-recovery techniques using checkpointing are discussed in [10]. It is assumed that one or more existing fault detection techniques in the literature (e.g., [7] or [1]) is used in the system to detect faulty components. The faults of interest are permanent faults that affect the processor of the PE or the memory module of the PE. Transient faults are not considered in this paper. For this study it is assumed that a faulty processor is unable to either compute or communicate with other PEs, but does not interfere with the operation of fault-free PEs. Furthermore, the local memory of a faulty processor is assumed to be corrupt, or inaccessible.

The failure of a processor local data bus or arithmetic logic unit would be examples of faults classified as processor faults. When a memory module is faulty, it is assumed that neither the local processor nor any other PE can reliably read data from or write data to the faulty memory module. The local processor is assumed to be fully operational in every other way. Such would be the case if there was a failure in the memory module refresh circuitry or a memory module I/O buffer, for example. A PE with a faulty processor or a faulty memory module will henceforth be referred to as a faulty PE, unless it is necessary to distinguish between the two fault types. When a fault is detected, the SCU determines and directs the proper recovery action after which processing continues from the last valid checkpoint.

3. RECOVERY OPTIONS AND COSTS

When a permanent fault occurs in a submachine A, three possible reconfiguration/recovery options are as follows.

1) Subdivide A into two equal-size system submachines, and use the one that is fault-free to complete the execution of the task.
2) Migrate the task to another submachine that is fault-free.
3) Redistribute the task programs and data among the fault-free PEs in A and complete the task using a modified algorithm that does not use the faulty PE.

These recovery options are discussed further in the following subsections.

If a PE memory module fault occurs, another reconfiguration/recovery option is possible. Using information from secondary storage or checkpoint data, the process that was executing on that PE can be loaded into the memory modules of other PEs in submachine A. Execution then continues as before, but with the processor associated with the faulty memory module accessing only remote memory. This option was considered in the research described here, but could not be included because of space limitations. Results of that work can be found in [25].

In addition to the costs of recovery discussed for each option, there is a time overhead of determining these recovery costs to select the best option for a given situation. However, this overhead is incurred prior to the initiation of any of these recovery schemes, so it is separate from the cost of recovery and is not included in the following subsections.

3.1. Task Completion on a Fault-Free Subdivision

When a PE fault occurs on a dynamically partitionable system, it can be avoided by subdividing the current submachine and completing the task on the fault-free subdivision. This process would proceed as follows. Once the PE fault has been located and the recovery point for the task has been determined, the SCU must make a determination about the partitionability of the current submachine. For this option, it is assumed that algorithmic constraints dictate that the subdivision is required to possess the same topological network properties as the original submachine. The recovery option of Section 3.3 is used when there is no need to preserve these properties. To create independent submachines with the same topological network properties as the original network, all submachines in a system possessing a multistage cube or hypercube interconnection network must have sizes that are a power of two [28, 30]. Thus, if a submachine can be partitioned (i.e., it is not of minimum size), it can be partitioned into two equal-size submachines. For a machine with $N = 2^n$ PEs, numbered from 0 to $N-1$, the numbers of all PEs in a submachine of size $K = 2^k$ agree in $n-k$ bit positions. These $n-k$ bits are called submachine bits. Without loss of generality, let these submachine bits be the low-order $n-k$ bits. The fault-free subdivision (half) B of this submachine A is composed of K/2 PEs. B's submachine bits are calculated by appending the complement of bit $n-k+1$ of the faulty PE number to the high-order end of submachine A's submachine bits. Thus, to determine the submachine bits of the fault-free subdivision (FFS) requires a constant amount of time T^{FFS}_{Check}.

The number of physical processors in the subdivision is half the number of physical processors in the original submachine. Therefore, the virtual processor ratio for the subdivision will double. The SC must determine the new mapping of virtual processors onto physical processors and coordinate the relocation of program code and data accordingly. T^{FFS}_{Map}, the time to perform this mapping, is computable by the SC and will depend on the implementation specifics of virtual processors in the system.

The next step is to move all the task program code and data onto the fault-free subdivision as per the virtual processor to physical processor mapping determined above. This will require T^{FFS}_{Tmsfr} time. The amount of code a task consists of can be easily determined during task compilation. However, the amount of data associated with a task can change dynamically during task execution. For this reason, part of the information saved during every checkpoint operation is the amount of checkpoint data saved. Because every PE stores its checkpoint data into a different PE within the same submachine, no data is lost when a PE becomes faulty. It is assumed that the interconnection network is used to transfer data. This transfer can be accomplished without conflicting with inter-PE messages from other submachines because of the partitioning properties of the network. The time to accomplish the data transfer can therefore be determined and will depend on the amount of data and the virtual processor to physical processor mapping.

In the SPMD (Single Program - Multiple Data) restriction of MIMD mode [7], every PE in the submachine has a copy of the same program. Thus, in SPMD mode, program code does not have to be transferred and T^{FFS}_{Tmsfr} will consist only of the time required to transfer the data. However, for SIMD and MIMD modes, some program code will have to be transferred.

The structure of SC/PE and inter-SC connections (if they exist) varies among approaches to multiple-SIMD architectures (e.g., CM-2 [32], MAP [22], PASM [29], TRAC[19]). Because of this, the time to do any needed transfer of SIMD programs when "moving" a SIMD task from one set of PEs to a subset or different set is highly machine dependent. Thus, in the discussions that follow, it will be assumed as an upper bound time requirement that the SIMD program is reloaded from secondary storage whenever a reconfiguration occurs.

For general MIMD mode tasks, program code will have to be moved from PEs not in the subdivision to those that are in the subdivision. Furthermore, depending on the mapping of virtual to physical processors, program code may have to be moved among PEs in the subdivision. The interconnection network is used to perform this transfer of program code and the time to do this can be determined in the same way as for the transfer of data across the network. However, the MIMD programs that resided on the faulty PE will have to be loaded from secondary storage (disk).

The time to transfer a SIMD or MIMD program from disk depends on the disk latency, the amount of code to be transferred, and the bandwidth of the disk communication channel. In such cases, T^{FFS}_{Tmsfr} is difficult to predict accurately because of variation in disk latency from one access to another. Therefore, an expected time for disk access will have to be used to determine T^{FFS}_{Tmsfr}.

The SCU must then perform whatever operating system functions are needed to establish the new system partitioning. This requires a system dependent time represented by T^{FFS}_{Part}. The time to complete the task execution on the subdivision is $T^{FFS}_{CmpExec}$ and will generally be greater than the time to complete the task execution on the original submachine. The difficulty of determining $T^{FFS}_{CmpExec}$ for this recovery option and those that follow is discussed in Sections 4 and 5. T^{FFS}_{Total}, the total time to reconfigure and complete a task on a subdivision of the original submachine is represented as follows.

$$T^{FFS}_{Total} = T^{FFS}_{Check} + T^{FFS}_{Map} + T^{FFS}_{Tmsfr} + T^{FFS}_{Part} + T^{FFS}_{CmpExec}$$

There are three main disadvantages of the subdivision recovery method. The first is that a subdivision of the current submachine may not exist, i.e., there is no way to further subdivide the current submachine. This is due to restrictions on minimum submachine size, which is usually associated with SIMD operation. The second disadvantage is similar and follows from this. There is a limit to how many faults can be tolerated in this way because of the physical limits to the number of times a machine can be partitioned. Finally, two types of performance degradation may occur. The first is task performance degradation. A task will generally require more time to complete on a subdivision of the original submachine. This is further discussed in Section 5. The second type of performance degradation is that of the system. After subdividing a submachine because of a fault, the entire subdivision containing the faulty PE becomes idle. In MIMD mode, the fault-free PEs of this subdivision can be utilized by the system because the PEs act independently of one another. However, in SIMD mode, the system must repeatedly partition the subdivision containing the fault until the fault is in a submachine of minimum size. Other tasks can be executed on the fault-free submachines created during this partitioning process. If the minimum size of the submachine containing the fault is greater than one, fault-free PEs become underutilized and thus contribute to a degradation in system performance.

3.2. Task Migration

It is likely that some tasks executing on a massively parallel processor will not use the entire machine and will therefore execute on an independent submachine formed by partitioning the machine. Reasons for partitioning include allowing multiple users (spatially), improving system utilization, and improving task execution time (some tasks can execute faster using fewer PEs) [28, 31]. Therefore, if one or more PE faults occur in the current (source) submachine P_s, the task can be migrated to a another (destination) submachine P_d, if one is available. It is assumed that P_s and P_d are of equal size. While this may not be a requirement for some systems, it is a reasonable assumption to make because P_s was originally of an appropriate size for the task. A brief analysis of the task migration costs discussed in [26] is provided below.

The first step to be made during the migration of a task from P_s to P_d is to decide to which P_d to migrate. When more than one P_d is available, the P_d that results in the least cost of migration should be chosen. Two factors that enter into this decision are the locations of P_s and P_d and the mapping of P_s PEs to P_d PEs. For a $N = 2^n$ PE machine, a O(n) time method for determining the mapping of PEs from P_s to P_d that minimizes the task migration (TM) time is provided in [27]. Thus, T^{TM}_{Map}, the time to choose P_d and to determine the optimal mapping of source PEs to destination PEs is a function of the log of the size of the machine and the number of destination submachines to be considered. The SCU performs this function, because it is responsible for allocating and deallocating submachines.

The next step in the task migration process is to transfer all necessary task information from P_s to P_d. The time to accomplish the data transfer depends on a combination of the amount of data to be transmitted, the location of source and destination submachines, the source PE to destination PE mapping, the use of the interconnection network by other tasks, the type of network, and system implementation details. It is assumed that the interconnection network is used to transfer data for the SIMD case, and programs and data for all other cases. It is also assumed that there is no interference during the transfer from other tasks because it is expected that that simultaneous task migrations would rarely occur. Furthermore, interference from normal inter-PE message traffic generated by a task(s) executing on a submachine(s) that the

task migrates through is additive in nature and very limited relative to the task migration traffic.

Let T^{TM}_{Tmsfr} be the time to transfer the program and data. For multistage cube interconnection networks and with the exception of SIMD and general MIMD program code, the time to perform this transfer is a linear function of the amount of data to be transferred and the number of network permutation settings required to accomplished the transfer. The amount of time to transfer the SIMD program code to P_d is machine dependent. In the worst case, the code will have to be loaded from secondary storage into the SC of P_d. For the general MIMD case, the program for the faulty PE must also be transferred from secondary storage. As discussed earlier, this access to secondary storage makes prediction of the transfer time difficult.

As before, $T^{TM}_{CmpExec}$ is the time to complete the task execution once the migration is complete. $T^{TM}_{CmpExec}$ will be equivalent to the time required to complete the task from the recovery point on the original submachine before the fault occurred. Thus, T^{TM}_{Total}, the time to migrate a task is as follows.

$$T^{TM}_{Total} = T^{TM}_{Map} + T^{TM}_{Tmsfr} + T^{TM}_{CmpExec}$$

The main advantages of task migration to another equal-size submachine are that no remapping of virtual processors to physical processors is required and once the migration is completed, the task will finish executing without any performance degradation. However, task migration is not possible if another submachine is not available. Also, the overhead to move a task may be significant compared to the task execution time remaining. As with the fault-free subdivision option, system degradation in the form of underutilization of fault-free PEs may occur for the PEs in P_s.

3.3. Task Redistribution

In many cases when a PE or PE memory fault occurs, it may be possible to redistribute the data and/or subtasks associated with the faulty PE to neighboring PEs within the same submachine, thus effectively removing the faulty PE from the computation. The ability to redistribute the task in such a way that the faulty PE is effectively removed from the computation depends on a number of factors including the algorithm, the mode of operation, and the interconnection network. Consider an algorithm that exhibits coarse grain parallelism and is implemented as a set of MIMD subtasks that require little inter-subtask communication. If a PE becomes faulty, the subtask that was executing on that PE can be distributed to one or more fault-free PEs. However, to minimize the execution time of the task, load balancing may be required. Work on load balancing on parallel and distributed systems [e.g., 21] has shown that the optimal load balancing solution depends on a number of factors including the task/subtask queuing model used, the precedence constraints involved, and on the tasks being executed. Thus, it is unlikely, using available technology, that a distribution solution can be found that works for all classes of problems. Furthermore, there is a trade-off between the time it takes to distribute the load and the savings in execution time that results.

Consider an SIMD machine with a mesh interconnection network. A faulty PE might require that all the data on all the PEs be considered in the redistribution to achieve optimal performance and preserve the near-neighbor communication pattern [34]. If a multistage cube network is used, it is possible that the network will not be able to support the required single-pass inter-PE communication permutations on a submachine where one or more PEs have been effectively removed (i.e., the permutations permitted are limited [17]). In general, there may be additional overhead incurred whether extra time is required to redistribute the data or extra time is needed to perform communication in a network where a needed permutation cannot be performed in one step due to a PE fault.

The first step in the task redistribution (TR) process is to determine how to accomplish the redistribution. In general, an equal load distribution (of data for SIMD/SPMD tasks and of both programs and data for MIMD tasks) among all of the fault-free PEs will result in the greatest efficiency for completion of the task. The time to make the redistribution, T^{TR}_{Map}, depends on the mode of operation, the algorithm mapping, and the current state of each subtask.

It is assumed that the SC has user-supplied knowledge about the algorithm mapping, which it uses to decide how to redistribute the task. Ideally, this knowledge would be compiler-supplied. However, while this may be possible for certain classes of problems, it is still an open problem in the general case.

The next step is to perform the subtask and/or data redistribution. T^{TR}_{Tmsfr} is the time required to accomplish the relocation of the program code and/or data. As in the previous two options, the time to accomplish this transfer depends on the mode of operation, the amount of data to be transferred, the location of source and destination PEs, the interconnection network, and system implementation details. Once again, for a MIMD task, the transfer of program code from secondary storage will add some unpredictability into the expected transfer time. The time to complete task execution after task redistribution, $T^{TR}_{CmpExec}$, is expected to be greater than the time to complete the task on the fault-free submachine and is discussed further in Sections 4 and 5. Thus, T^{TR}_{Total}, the time to redistribute a task among the fault-free PEs of a submachine and complete execution of the task is as follows.

$$T^{TR}_{Total} = T^{TR}_{Map} + T^{TR}_{Tmsfr} + T^{TR}_{CmpExec}$$

An advantage of the task redistribution option is that all the fault-free PEs in the submachine can be utilized by the task, while with the fault-free subdivision option, only half the number of original PEs are utilized. A disadvantage is that to allow data and subtasks to be redistributed with a minimum of effort, the task must be

coded and/or compiled considering fault tolerance and reconfiguration. As an example, linked lists should be favored over indexed arrays when random access to the list elements is infrequent, because the code does not have to know the number of elements in the structure and does not have to be modified when the data is redistributed. Of the three options for reconfiguration discussed, the task redistribution option is by far the most difficult to quantify. However, if the task redistribution option is to be considered, the information discussed in this section is needed to choose the best alternative.

4. DATA-INDEPENDENT MODEL

One recovery cost common to all the options presented is the remaining execution time. However, the time required to complete execution of a task varies with the recovery option. For example, a task that has nearly completed executin$_g$ may complete much earlier on a subdivision than it would if it were migrated to another submachine, especially if a high overhead migration cost is incurred. Alternatively, if a significant amount of computation remains, migrating the task may result in an earlier task completion time. Thus, the remaining task execution time becomes important.

Tasks can be divided into two categories based on execution time. These are tasks with data-independent execution times and tasks with data-dependent execution times. A task with a data-independent execution time does not depend on input data to make branching decisions. Thus, the number of times any branch in the task program code is taken can be determined by a compiler during program compilation. A task with a data-dependent execution time, on the other hand, has branch decisions that are based on data that is known only at run time. The execution time of such tasks is considered in the next section.

A model is presented in this section for determining the execution-time costs of data-independent-execution-time tasks. This model differs from other models in that it can accommodate a logically shared or hybrid memory system. Such an execution-time model must necessarily include detail about the amount of time spent making local and remote memory accesses. Both SIMD and MIMD versions of the model are presented. Because SPMD is a subset of MIMD, it is accommodated by the MIMD model. Mixed-mode operation execution-time prediction can be accomplished by extending the model to account for switching between modes.

4.1. SIMD Task Execution Time

In SIMD mode, it is assumed that instructions to be executed by the submachine PEs are loaded into an automatic broadcast queue by the SC. This leaves the SC free to perform computations of its own (e.g., the manipulation of loop index variables, PE-common array index calculations). Thus, the execution time for an SIMD task includes the SC execution time plus the PE execution time minus any overlap between the two. For a comprehensive model for determining the amount of SC/PE overlap in an SIMD task see [16]. Let T_{SC} be the time the SC spends performing its calculations, T_{OL} be the total SC/PE overlap, and $T_{PE}^{P(i)}$ be the execution time of instruction i on processor P. For a disabled PE, $T_{PE}^{P(i)} = 0$. In SIMD mode, a new instruction is broadcast to the PEs by the SC only after the current instruction's execution has been completed (or initiated, if prefetching is used). Therefore, T_{SIMD}, the overall time for a SIMD task consisting of the execution of I instructions on Q PEs, $0 \le P \le Q-1$, is

$$T_{SIMD} = T_{SC} - T_{OL} + \sum_{i=1}^{I} \max_{P} \left(T_{PE}^{P(i)} \right).$$

Let t_L be the time required for a PE to make a single access to its local memory. It is assumed that any such access takes a constant amount of time. Similarly, let t_R be the average time required to make an access to a remote (non-local) memory location.

For the architectures under consideration, a remote memory reference requires use of the interconnection network. For a read of remote memory, an address must traverse the network to the remote memory and the memory contents must traverse the network to the source of the read request. For a remote memory write, an address and word to be written must be sent to the remote memory location. Generally, an acknowledgment of some type is returned by the remote network interface. The time required for an address or data word to traverse the network can be divided into two components. These are transfer time and overhead. The transfer time is the actual propagation time required assuming no contention in the network. Transfer time is a constant for networks that have equidistant paths between any source and destination pair (e.g., multistage cube network), but may vary for other networks (e.g., hypercube). The overhead is time spent to establish a path through the network and the time spent waiting due to network contention or contention at the remote memory. In general, the determination of overhead is a difficult problem that depends on the network implementation details and the message traffic. Thus, t_R is only an estimated average value; calculating an exact time for each transfer would require machine and application-dependent analysis. Research on detailed modeling of networks has been the topic of entire papers [e.g., 11], and is beyond the scope of this work.

Here, for the sake of simplicity, remote memory reads and writes are assumed to take the same amount of time. The model could be extended to include different costs for the two types of references.

The number of local and remote memory references made by PE P during instruction i (when no PE is faulty) is denoted by $M_L^{P(i)}$ and $M_R^{P(i)}$, respectively. A compiler could determine these values for every instruction in a task as well as the amount of time spent by PE P on instruction i doing non-memory reference work ("computation time"), $T_C^{P(i)}$. For a disabled PE, $M_L^{P(i)} = 0$, $M_R^{P(i)} = 0$, and $T_C^{P(i)} = 0$. Thus, the total time spent by PE P

on instruction i is given by

$$t_L M_L^{P(i)} + t_R M_R^{P(i)} + T_C^{P(i)}.$$

Therefore, the new SIMD execution-time model becomes

$$T_{SIMD} = T_{SC} - T_{OL} + \sum_{i=1}^{I} \max_{P} \left[t_L M_L^{P(i)} + t_R M_R^{P(i)} + T_C^{P(i)} \right].$$

The model for determining SIMD task execution time presented above can be applied in cases where the exact sequence of instructions to be executed is known at compile time. However, for most programs the sequence of instructions executed is data dependent. For such programs it is not possible to determine an execution time during compilation. Furthermore, it is not practical for the SC to determine the amount of execution time left when a fault occurs, because it would potentially have to simulate the execution of every instruction on every PE to determine when the task has completed. In cases where empirical data is available on the execution time of a particular task, it is possible to determine an estimate of the execution time remaining. Such an estimate is discussed in the next section. For data-dependent-execution-time tasks where there is little empirical data, there is no practical way of determining an execution time. In such cases, the reconfiguration decision may have to be made without benefit of execution-time knowledge.

4.2. MIMD Task Execution Time

It is assumed in the following model that an instruction cycle is composed of a fetch phase and an execute phase. Because each PE may execute a different program in MIMD mode, let I(P) denote the number of instructions that PE P must execute to complete a task and let $T_{PE}^{P(i)}$ be the time for PE P to execute its i^{th} instruction. Then, T_{MIMD}, the execution time for a task in MIMD mode is

$$T_{MIMD} = \max_{P} \left[\sum_{i=1}^{I(P)} T_{PE}^{P(i)} \right].$$

This "max of sums" for MIMD versus "sum of max's" for SIMD has been discussed in another context in [3].

Let $W^{P(i)}$ represent the number of words that PE P must fetch from local memory to read in instruction i. As before, the number of local and remote memory references made by PE P during its i^{th} instruction is denoted by $M_L^{P(i)}$ and $M_R^{P(i)}$ respectively. A compiler could determine these values for every instruction in a task as well as $T_C^{P(i)}$, the amount of time spent by PE P on its i^{th} instruction doing non-memory-reference work. Thus, the MIMD task execution time becomes

$$T_{MIMD} = \max_{P} \left[\sum_{i=1}^{I(P)} \left(t_L \left[M_L^{P(i)} + W^{P(i)} \right] + t_R M_R^{P(i)} + T_C^{P(i)} \right) \right].$$

As with the SIMD execution-time model, the equation above is useful only when the exact sequence of instructions for every MIMD subtask is known. The MIMD execution-time equation given does not account for conflicts in the interconnection network, so the execution time derived from the equation will be a minimum execution time.

5. DATA-DEPENDENT MODEL

Most tasks have a data-dependent execution time, making the execution-time model presented in the preceding section inapplicable. However, in a production environment where a task is executed repeatedly on various sets of data (e.g., image processing of satellite pictures), empirical studies can be performed to derive an estimated execution time for a task. For these cases it is possible to develop a model that will predict the expected execution time remaining for a task that is recovering from a fault.

5.1. Execution Time on a Different Submachine

Once a task has been migrated from a submachine containing a faulty PE to a fault-free submachine of equal size, the time to complete the task execution will be the same as it would have been on the original fault-free submachine. An estimate of this expected completion time is now derived.

Let $\hat{T}_{Exec}$ be a discrete random variable that represents the execution time for a specific task. T_{Exec} is always positive because a negative execution time is not possible. $f(T_{Exec})$ is the discrete density function [23] of $\hat{T}_{Exec}$. If $\hat{T}_{Exec}$ takes the value x_i with probability p_i, and $\delta(x)$ is the impulse function such that $\delta(x - x_i)$ has value one at $x = x_i$ and value zero elsewhere, then

$$f(T_{Exec}) = \sum_i p_i \delta(T_{Exec} - x_i).$$

By definition, the expected value of $\hat{T}_{Exec}$ is

$$E\left\{\hat{T}_{Exec}\right\} = \int_0^{\infty} T_{Exec} f(T_{Exec})\, dT_{Exec} = \eta_{exec}.$$

It is assumed that the amount of execution time spent on a task prior to a checkpoint is stored with that checkpoint. If a recovering task is to proceed from a checkpoint and the execution time stored with that checkpoint is τ, $T_{CmpExec}$, the expected amount of time required to complete task execution is

$$T_{CmpExec} = \eta_{exec} - \tau.$$

$T_{CmpExec}$ assumes that the submachine size remains the same and that all the PEs in the submachine are fault-free.

5.2. Execution Time on a Fault-Free Submachine

Here, the completion time of a task that completes execution on a subdivision that is half the size of the original submachine is considered. Let T_{2x} be the expected execution time of a task on a submachine consisting of 2x PEs, and let T_x be the expected execution time of the

same task remapped onto one of the two equal-sized subdivisions of the original submachine. If the average time for an inter-PE transfer remains the same in either case,

$$T_{2x} \geq \frac{1}{2} T_x.$$

The inequality becomes an equality when the mapping of the task from the subdivision onto the 2x PE submachine is optimal.

This equation can be extended to submachines of arbitrary size as follows. Let S_{Old} be the number of PEs in the old (original) submachine and let S_{New} be the number of PEs in the new submachine (subdivision). Then, the expected remaining execution time on the subdivision is bounded by

$$T_{CmpExec} \leq \frac{S_{Old}}{S_{New}} \left[\eta_{exec} - \tau \right].$$

A more accurate remaining execution time estimate can be obtained if empirical data is available to determine expected task execution times for the submachine sizes of interest. Then, the estimated task execution time becomes a function of the submachine size and the estimate of the remaining execution time becomes

$$T_{CmpExec} = \eta_{exec}(S_{New}) \left[1 - \frac{\tau}{\eta_{exec}(S_{Old})} \right].$$

5.3. Execution Time After Task Redistribution

An execution-time estimate for the task redistribution recovery option is more difficult than for the previous options. Consider a task executing on a submachine of size S_{Old} in MIMD mode. If a PE becomes faulty and its subtasks are distributed equally to the $S_{New} = S_{Old} - 1$ fault-free PEs in the submachine, the remaining execution time is bounded as follows.

$$T_{CmpExec} \leq \frac{S_{Old}}{S_{Old} - 1} \left[\eta_{exec} - \tau \right]$$

Consider the case where the faulty PE's subtasks cannot be distributed equally among the fault-free PEs. In the worst case, all the faulty PE's subtasks would be assigned to one PE. In this case, the remaining execution time could be twice that of the remaining execution time on a fault-free submachine. The degree to which the performance of the system on a task is degraded is a function of the system (α), the algorithm implementation (β), and the number and location of faulty PEs (γ). Let $d(\alpha,\beta,\gamma)$ be defined as the amount of performance degradation: $1 < d(\alpha,\beta,\gamma) \leq 2$. The amount of performance degradation can be estimated from empirical data or from user provided information about the algorithm implementation. The remaining execution time then becomes

$$T_{CmpExec} = d(\alpha,\beta,\gamma) \left[\eta_{exec} - \tau \right].$$

Using the execution-time models presented in this section, one can arrive at an estimate of the remaining execution time for a task after recovery from a fault. The remaining execution time for tasks with data-dependent execution times for which no empirical data is available cannot be estimated reliably. In these cases, the user must be required to supply an estimate prior to execution time if a comparison of recovery options is to be performed.

An alternative method for determining remaining execution time centers around the use of an automatic complexity evaluator such as that presented in [18]. The approach here is to attempt to generate a nonrecursive function, prior to run time, that can be solved at run time to determine the asymptotic time-complexity behavior of a program. This approach is applicable to a wide range of programs, but it is not always successful in generating a nonrecursive function. Further study is needed to see how it can be applied here.

6. CONCLUSIONS

The viability of a quantitative model of system reconfiguration due to a PE or PE memory module fault was examined. Three fault recovery options were discussed and the parameters required to determine their respective costs were identified. It has been shown that given the present technology, collecting precise values for some of these parameters is very difficult. As an example of this, two models for determining the remaining execution time of a task after reconfiguration to tolerate a fault were presented. The first is applicable to tasks with data-independent execution times and relies on compile-time analysis of the task. The second execution-time model is for use with tasks with data-dependent execution times that are production oriented. This model uses empirically derived or user suplied expected execution times to estimate the remaining execution time for tasks.

The model presented in the preceding sections can be used to derive some heuristics to aid in the decision of which option to choose. Consider a SIMD task for which every PE in a subdivision of the submachine checkpoints to a PE in the other subdivision in such a way that no two PEs checkpoint to the same PE. Then, either subdivision will contain all the data needed to recover from a PE fault and no data will have to be moved during the recovery if the subdivision option is used (although a better mapping may exist). Let D_{ChkPnt} be the amount of data checkpointed per PE during the last checkpoint operation and let t_d be the expected time to transfer one data item across the interconnection network. If $D_{ChkPnt} \cdot t_d < (\eta_{exec} - \tau)$, the task migration option should not be considered because it would take longer than completing the task on a subdivision.

It is possible that the overhead required to determine the best fault-recovery option may be greater than the possible benefit of selecting the best option. Let T_{Decide} be the expected time to decide on the best option. For an SIMD task, if $2(\eta_{exec} - \tau) < T_{Decide}$, the subdivision

option can be used to complete the task before a better option (if one exists) can be selected. The subdivision option is chosen because there is no possibility of interference with tasks on other submachines and no algorithm dependent redistribution of data (and associated overhead) is needed.

For MIMD and SPMD tasks, if all the subtasks of the faulty PE are redistributed to one fault-free PE, the task will complete in approximately the same amount of time as it would on a subdivision. When the subtasks of the faulty PE are redistributed to more than one fault-free PE, the redistribution option will be faster than the subdivision option. Thus, as a general heuristic, there is no need to consider the subdivision option for MIMD and SPMD tasks.

For production environment tasks it may be possible to collect empirical data on expected completion time for a task as a function of the time the task spent executing before the fault and the recovery method used (the fault location may also have to be considered). Then, the choice of recovery options becomes simply a matter of chosing the option with the smallest completion time given the point at which the fault occurred. Of course, for the task migration option it may be impossible to predict the amount of interference possible from tasks executing on other submachines.

In conclusion, using the model presented here, it may be possible to choose an "optimal" reconfiguration strategy for some classes of tasks executing on a given machine. However, much work remains before a quantitative model of system reconfiguration will be practical for all classes of tasks. Analysis of such models helps designers of fault tolerant systems determine requirements for compilers and operating systems to provide needed input parameters for the model. It also serves to identify areas where new technology is needed.

Plans for future work include the validation of the model presented here by experimentation and/or simulation.

Acknowledgments: The authors gratefully acknowledge useful discussions with James Armstrong, Hasan Cam, and Peter Wahle. The authors also wish to thank the anonymous referees for their helpful suggestions.

REFERENCES

[1] V. Balasubramanian and P. Banerjee, "CRAFT: Compiler-assisted algorithm-based fault tolerance in distributed memory multiprocessors," *1991 Int'l Conf. on Parallel Processing*, v. I, Aug. 1991, pp. 505-508.

[2] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Trans. Comput.*, v. C-31, May 1982, pp. 377-384.

[3] T. B. Berg, S. D. Kim, and H. J. Siegel, "Limitations imposed on mixed-mode performance of optimized phases due to temporal juxtaposition," *J. Parallel Distrib. Comput.*, Special Issue on Massively Parallel Computation, v. 13, Oct. 1991, pp. 154-169.

[4] T. Blank, "The MasPar MP-1 architecture," *IEEE Compcon*, Feb. 1990, pp. 20-24.

[5] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 531-540.

[6] A. T. Dahbura, G. M. Masson, and C. Yang, "Self-implicating structures for diagnosable systems," *IEEE Trans. Comput.*, v. C-33, Aug. 1985, pp. 718-723.

[7] F. Darema-Rodgers, D. A. George, V. A. Norton, and G. F. Pfister, *Environment and System Interface for VM/EPEX*, Research Report RC11381 (#51260), IBM T. J. Watson Research Center, 1985.

[8] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting," *J. Parallel Distrib. Comput.*, v. 11, Mar. 1991, pp. 239-251.

[9] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, v. 54, Dec. 1966, pp. 1901-1909.

[10] T. M. Frazier and Y. Tamir, "Application-transparent error-recovery techniques for multicomputers," *Fourth Conf. on Hypercubes, Concur. Comput., & Appl.*, Mar. 1989, pp. 103-108.

[11] P. G. Harrison, "Analytic Models for Multistage Interconnection Networks," *J. Parallel Distrib. Comput.*, Special Issue on Modeling of Parallel Computers, v. 12, Aug. 1991, pp. 357-369.

[12] J. Hastad, T. Leighton, and M. Newman, "Reconfiguring a hypercube in the presence of faults," *19th ACM Symp. Theory Comput.*, 1987, pp. 274-284.

[13] J. P. Hayes, T. N. Mudge, Q. F. Stout, and S. Colley, "Architecture of a hypercube supercomputer," *1986 Int'l Conf. on Parallel Processing*, Aug. 1986, pp. 653-660.

[14] D. J. Hunt, "AMT DAP - a processor array in a workstation environment," *Comput. Sys. Sci. & Engr.*, Vol. 4, No. 2, April 1989, pp. 107-114.

[15] Intel Corporation, *A New Direction in Scientific Computing*, Order # 28009-001, Intel Corp., 1985.

[16] S. D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling overlapped operation between the control unit and processing elements in an SIMD machine," *J. Parallel Distrib. Comput.*, Special Issue on Modeling of Parallel Computers, v. 12 Aug. 1991, pp. 329-342.

[17] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, v. C-24, Dec. 1975, pp. 1145-1155.

[18] D. Le Métayer, "ACE: An Automatic Complexity Evaluator," *ACM Trans. Prog. Lang. & Sys.*, v. 10, Apr. 1988, pp. 248-266.

[19] G. J. Lipovski, and M. Malek *Parallel Computing: Theory and Comparisons*, Wiley, New York, 1987.

[20] M. Livingston and Q. F. Stout, "Fault tolerance of allocation schemes in massively parallel computers," *Frontiers '88: The 2nd Symp. on the Frontiers of Massively Parallel Comput.*, Oct. 1988, pp. 491-494.

[21] W. G. Nation, A. A. Maciejewski, and H. J. Siegel, "Exploiting concurrency among tasks in partitionable parallel supercomputers," *1992 Int'l Parallel Processing Symp.*, Mar. 1992, pp. 30-38.

[22] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Fourth Anl. Symp. on Comput. Architecture*, Mar. 1977, pp. 147-152.

[23] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, Second Edition*, McGraw-Hill, New York, NY, 1984, p. 71.

[24] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," *1985 Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 764-771.

[25] G. Saghi, H. J. Siegel, and J. A. B. Fortes, *On a Quantitative Model of System Reconfiguration Due to a Fault*, Tech. Rep. in preparation, EE School, Purdue.

[26] T. Schwederski, H. J. Siegel, and T. L. Casavant, "A model of task migration in partitionable parallel processing systems," *Frontiers '88: The 2nd Symp. on the Frontiers of Massively Parallel Comput.*, Oct. 1988, pp. 211-214.

[27] T. Schwederski, H. J. Siegel, and T. L. Casavant, "Optimizing task migration transfers using multistage cube network," *1990 Int'l Conf. on Parallel Processing*, v. I, Aug., 1990, pp. 51-58.

[28] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, v. C-29, Sep. 1980, pp. 791-801.

[29] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, pp. 387-407.

[30] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition*, McGraw-Hill, New York, NY, 1990.

[31] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems," *Computer*, Special Issue on Parallel Processing for Computer Vision and Image Understanding, v. 25, Feb. 1992, pp. 54-63.

[32] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, v. 21, Aug. 1988, pp. 26-38.

[33] M. U. Uyar and A. P. Reeves, "Fault reconfiguration in a distributed MIMD environment with a multistage network," *1985 Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 798-806.

[34] M. U. Uyar and A. P. Reeves, "Dynamic fault reconfiguration in a mesh-connected MIMD environment," *IEEE Trans. Comput.*, v. C-37, Oct. 1988, pp. 1191-1205.

REFERENCE NO. 19

Cam, H. and Fortes, J. A. B., "Frames: A Simple Characterization of Permutations realized by Frequently Used Networks," Tech. Rpt. TR-EE-92-32, School of Electrical Engineering, Purdue University, July 1992, 44 pages.

Note - This report proposes a simple characterization of permutation capabilities of interconnection networks. The proposed approach makes it computationally easy to detect whether a particular interconnection pattern (i.e., configuration) can be implemented by a network. Submitted for publication in IEEE Transactions on Computers.

FRAMES: A SIMPLE CHARACTERIZATION OF PERMUTATIONS REALIZED BY FREQUENTLY USED NETWORKS

HASAN CAM
JOSE´ A.B. FORTES

TR-EE 92-32
JULY 1992

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

FRAMES: A SIMPLE CHARACTERIZATION OF PERMUTATIONS REALIZED BY FREQUENTLY USED NETWORKS

Hasan Çam and José A. B. Fortes

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285

ABSTRACT

Rearrangeable multistage interconnection networks such as the Benes network realize any permutation, yet their routing algorithms are not cost-effective. On the other hand, non-rearrangeable networks can have inexpensive routing algorithms, but no simple technique exists to characterize *all* the permutations realized on these networks. This paper introduces the concept of frame and shows how it can be used to characterize all the permutations realized on various multistage interconnection networks. They include any subnetwork of the Benes network, the class of networks that are topologically equivalent to the baseline network, and cascaded baseline and shuffle-exchange networks. The question of how the addition of a stage to any of these networks affects the type of permutations realized by the network is precisely answered. Also, of interest from a theoretical standpoint, a new simple proof is provided for the rearrangeability of the Benes network.

Index Terms— Multistage interconnection network, permutations, rearrangeability, topological equivalence, balanced matrices, frames.

This research was supported in part by the Office of Naval Research under contract No. 00014-90-J-1483 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-88-k-0723.

List of Symbols

IN: interconnection network.

IP: interconnection pattern.

IP_{in}: interconnection pattern formed by input links.

IP_{out}: interconnection pattern formed by output links.

SB: switching box; switch.

BE: baseline network; see Definition II.3.

RB: reverse baseline network; see Definition II.3.

SE: shuffle-exchange network; see Definition II.3.

SE^{-1}: inverse shuffle-exchange network; see Definition II.3.

BS: Benes network; see Definition II.3.

CS: Clos network; see Definition II.3.

N: number of inputs/outputs of a network.

n: $\log_2 N$.

$F^{\alpha}_{1:k}$: the standard α-type frame with k columns; see Definition III.2.

$F^{\alpha}_{1:k}$: an α-type frame with k columns; see Definition III.3.

$F^{*}_{1:k}$: the universal frame with k columns; see Definition III.6.

I: the identity permutation matrix; see Definition II.1.

R: the reverse permutation matrix; see Definition II.1.

r: the reverse permutation represented by R.

$A_{N\times k}$: matrix A with N rows and k columns.

$A_{N\times k}(i)$: the ith row of matrix A.

γ: a permutation on the set $\{0,1,\ldots,N-1\}$; see Definition III.2.

β: a mapping of the set $\{1,2,\ldots,k\}$ into $\{1,2,\ldots,n\}$; see Definition III.2.

P: a tuple of partitions; see Definition III.2.

I. INTRODUCTION

Interconnection networks are utilized to provide communication among processing elements and/or memory modules. Network performance significantly affects the overall cost and performance of a computational system because processors may spend a considerable amount of time in processor-processor and/or processor-memory communication. Therefore, it is important to know exactly the interconnection patterns that can be implemented by a network. In particular, it is desirable to know what *permutations* can be realized because parallel algorithms often require permutation-type data transfers. This paper presents a simple and easily understandable characterization of the permutations realized by any network with $N=2^n$ inputs that is topologically equivalent to one of the following networks: first k stages, $1 \le k \le n$, of the reverse baseline network, the last $n+k-1$ stages of Benes network [7], or a cascade of baseline [11] and k-stage shuffle-exchange [1,5] networks. The proposed characterizations are based on the notion of *"frame"* (introduced in this paper), balanced matrices [2] and graph theory [3,4].

The effectiveness of any interconnection network depends on factors such as the efficiency of the routing algorithm, the number and type of permutations it realizes, and the actual hardware implementation of the network. On one hand, rearrangeable multistage interconnection networks such as Benes and $\Omega\Omega^{-1}$ (the $\Omega\Omega^{-1}$ is a cascade of omega and inverse omega [1]) can realize any permutation. However, there are no known efficient routing algorithms to allow dynamic configuration in an environment where the switching permutations change rapidly. On the other hand, some networks such as baseline and omega have efficient routing algorithms and small propagation delays, but cannot realize many permutations. In these cases, it is important to know which permutations are realizable and this is possible by using the results of this paper.

Different approaches have been proposed in the literature to circumvent inefficient routing algorithms. One approach is to determine certain types of permutations that occur more frequently than others in a parallel processing environment. Such permutations have been classified by Lenfant [23] into five families. In order to implement these permutations on the Benes network with a small propagation delay, Lenfant proposed a specialized routing algorithm for each family. A permutation that fails to be in one of these families still is routed using an inefficient routing algorithm. To increase the number of the families of permutations that can be realized by a network, Youssef and Arden [22] introduced an $O(\log^2 N)$ routing algorithm which sets the $(r \times r)$ crossbar switches of the first stage of 3-stage Benes networks with $N=r^2$ inputs to a fixed configuration and acts exactly like a self-routing algorithm in setting the remaining switches. Another approach is to provide self-routing algorithms for realizing some classes of permutations in various multistage interconnection networks such as Benes, $2n$-stage shuffle-exchange. Nassimi and Sahni [24] presented simple self-

routing algorithms to realize some important permutations in Benes networks. Raghavendra and Boppana [25] proposed self-routing algorithms to realize the class of linear permutations on Benes and $2n$-stage shuffle-exchange networks.

Although a large number of multistage interconnection networks are extensively studied, there is a relatively small number of basic designs for their underlying topologies. Especially, Benes networks and six topologically equivalent networks, namely, omega, flip, indirect binary cube, modified data manipulator, baseline and reverse baseline have been investigated in depth and are frequently used in research studies and real systems. Characterizations of the topologies of these networks are given in [9,26,27]. However, to our knowledge, the characterization of the permutations performed by *these and other* networks is done for the first time in this paper. One exception is the work of Lee [10] which characterizes the permutations realized by the inverse omega network in terms of residue classes.

The rest of the paper is organized as follows. Basic definitions and notations used throughout the paper are presented in Section II. Also included in this section is a motivational example for the concept of frame. In Section III, this concept, illustrations of many different frames, notation and terminology are introduced. Permutations realized by the k-stage reverse-baseline, $1 \le k \le n$, and the networks which are topologically equivalent to it are characterized in Section IV. In Section V, the permutations realized by a cascade of reverse baseline and the k-stage shuffle-exchange networks are identified. These cases show how frames can be used to characterize the permutations of some relatively complex networks with more than n stages. Section VI provides new proofs for the rearrangeability of the three-stage Clos and Benes networks. Permutations realized by the last $n+k-1$ stages of Benes network are identified in Section VII. This characterization illustrates how frames can be used to understand why a network is rearrangeable. Section VIII concludes the paper. The Appendix (Section IX) contains the proofs of most of theorems and lemmata in the paper.

II. BASIC DEFINITIONS AND A MOTIVATIONAL EXAMPLE

Throughout this paper, matrices are denoted by single capital letters and columns of a matrix are represented by the lower case of the capital letter denoting that matrix. Matrix A having N rows and k columns is denoted by $A_{N\times k}$. Given a matrix, e.g. $A_{N\times k}$, the jth column is denoted by a_j, $1 \le j \le k$. To be able to refer to a set of specific columns of a matrix, the notation $A_{x:y}$ is used to denote the submatrix that contains those columns of A whose indices are x, $x+1$, $\cdots$,y, where $1 \le x \le y$; if x happens to be greater than y, then $A_{x:y}$ refers to a nil matrix, unless stated otherwise. If $x=y$, then $A_{x:y}$ refers to a single column a_x. Unless specifically stated, the number of the rows of a matrix $A_{x:y}$ is assumed to be equal to N. $A_{N\times n}(i)$ refers to the ith row of the matrix

$A_{N\times n}$, where $0 \le i \le N-1$. A column vector of N entries of which half are 0's and the other half are 1's is called a *column permutation*. Unless otherwise stated, any column of any matrix in this paper is a column permutation. The binary representation of a positive integer $0 \le b \le N-1$ is $(b_1 b_2 \cdots b_n)$ such that $b = b_1 .2^{n-1} + b_2 .2^{n-2} + \cdots + b_n .2^0$.

A *permutation* on a set X is a bijection of X onto itself. A permutation f permutes the ordered list $0, 1, \cdots, N-1$ into $f(0), f(1), \cdots, f(N-1)$. A cyclic notation [20,21] can be used to represent a permutation as the product of cycles, where a cycle $(c_0\, c_1\, c_2 \cdots c_{k-1}\, c_k)$ means $f(c_0) = c_1$, $f(c_1) = c_2$, $\cdots$, $f(c_{k-1}) = c_k$, and $f(c_k) = c_0$. The composition of several permutations $f_1 . f_2 \cdots f_k$ is evaluated from left to right, i.e., it maps i into $f_k(\cdots(f_2(f_1(i)))\cdots)$.

Definition II.1. (Permutation matrix, identity permutation matrix, reverse permutation matrix): A permutation h can be represented by a $N \times n$ binary matrix called *permutation matrix*, H, such that its ith row, $H_{N\times n}(i)$, is the binary representation of the integer $h(i)$. The *identity permutation matrix* denoted by $I_{N\times n}$ is the matrix whose ith row is the binary representation of i (this is called "standard matrix" in [12]). The *reverse permutation matrix*, denoted $R_{N\times n}$, is the matrix whose jth column is the $(n+1-j)$th column of $I_{N\times n}$.

For instance, the identity permutation matrix $I_{8\times 3}$, the reverse permutation matrix $R_{8\times 3}$ and a permutation matrix $E_{8\times 3}$ are shown below:

$$I_{8\times 3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad R_{8\times 3} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad E_{8\times 3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Clearly, there is a one-to-one correspondence between permutations and permutation matrices. For instance, $R_{8\times 3}$ represents the permutation r:

$$r = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 2 & 6 & 1 & 5 & 3 & 7 \end{bmatrix}.$$

Using the cyclic notation, r is represented by $r = (0)(1\ 4)(2)(3\ 6)(5)(7)$.

II.1. Networks

In the terminology used in this paper, a k-stage interconnection network (IN) consists of k columns of switching boxes (SBs), each followed and preceded by links which form *interconnection patterns (IPs)* as shown in Figure II.1. The *IPs* formed by the input and output links are denoted by IP_{in} and IP_{out}, respectively. Thus, an IN contains $(k+1)$ interconnection patterns labeled IP_{in}, IP_1, $IP_2, \ldots, IP_{k-1}$, IP_{out}. A column of IN contains $N/2$ (2×2) SBs, each of which can be set either straight or cross. Figures II.2, II.3, II.4, II.5, and III.6 show several networks considered in this paper for $N=16$, namely, reverse baseline, baseline, Benes, the 4-stage shuffle-exchange (SE), and the 4-stage inverse SE. If some networks are placed in parallel to form a new IN, then the IN is said to be a "pile of networks". Unless otherwise stated, any IN is assumed to have N inputs/outputs and its stages are labeled from left to right starting with 1. Network stages are defined below and illustrated in the figures.

Definition II.2. (Stages of reverse baseline, baseline, Benes, SE, and inverse SE networks): With one exception, a stage in the reverse baseline and SE networks consists of a connection pattern and the following column of SBs. The exception is the rightmost stage (i.e., the output stage) which consists of the last column of SBs and both the preceding and succeeding connection patterns. Stages are labeled from left to right in ascending order starting with 1. In the baseline network the kth stage corresponds to the $(n-k+1)$th stage of the reverse baseline network. (Notice that both the reverse baseline and the baseline can have at most n stages, by definition). In the inverse SE network with m stages, its kth stage corresponds to the $(m-k+1)$th stage of the m-stage SE network. In this paper, Benes network is considered as being composed of the first $n-1$ stages of the n-stage baseline followed by the n-stage reverse baseline. (It could also be considered as being composed of the n-stage baseline followed by the last $n-1$ stages of the n-stage reverse baseline). Therefore, the stages of Benes network are labeled according to the labeling rules of the baseline and the reverse baseline.

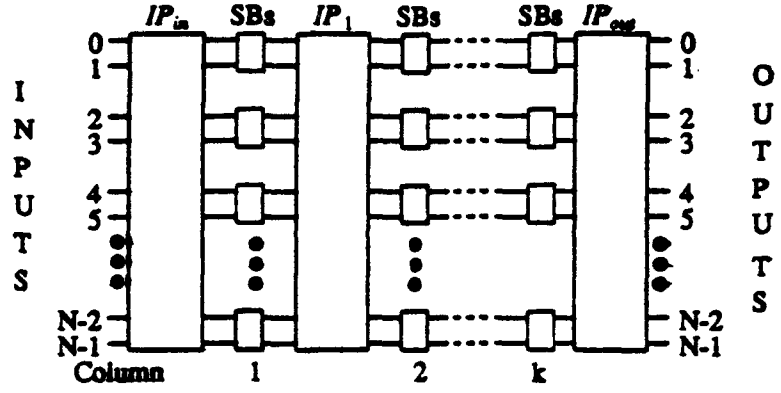


Figure II.1. An IN with (2×2) SBs and interconnection patterns shown as large boxes.

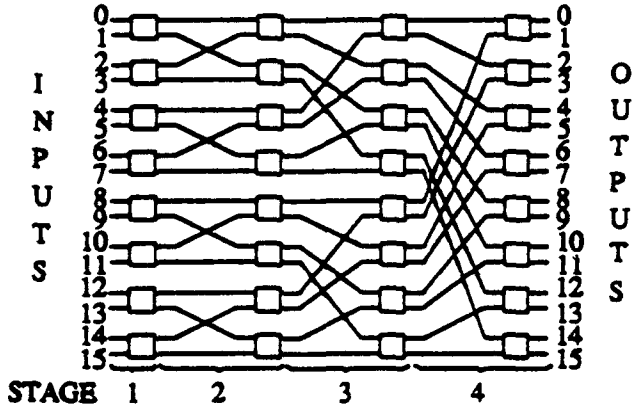


Figure II.2. The 4-stage reverse baseline network with 16 inputs/outputs.

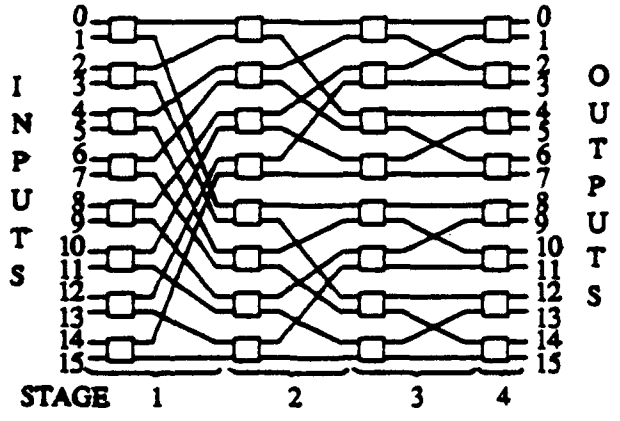


Figure II.3. The 4-stage baseline network with 16 inputs/outputs.

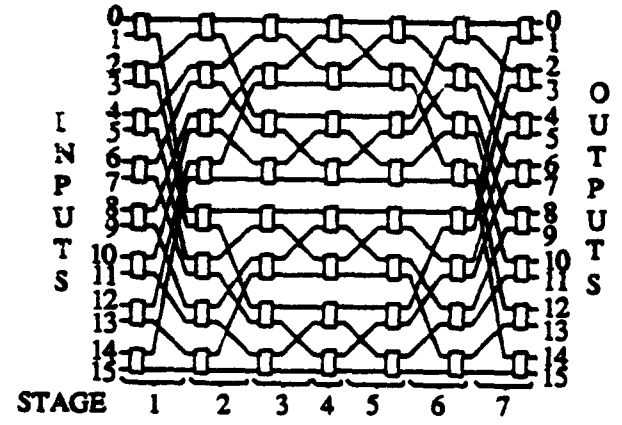


Figure II.4. Benes network with 16 inputs/outputs.

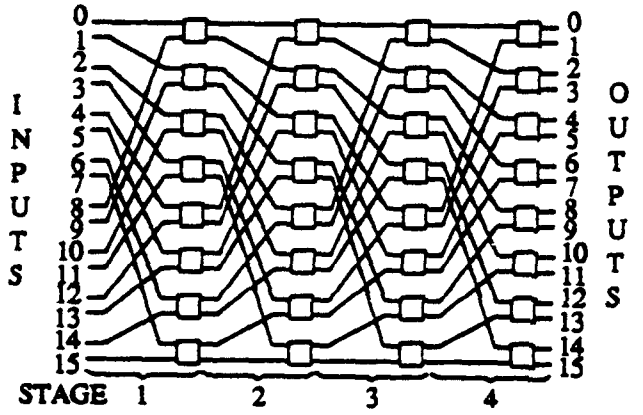


Figure II.5. The omega network (i.e., the 4-stage SE) with 16 inputs/outputs.

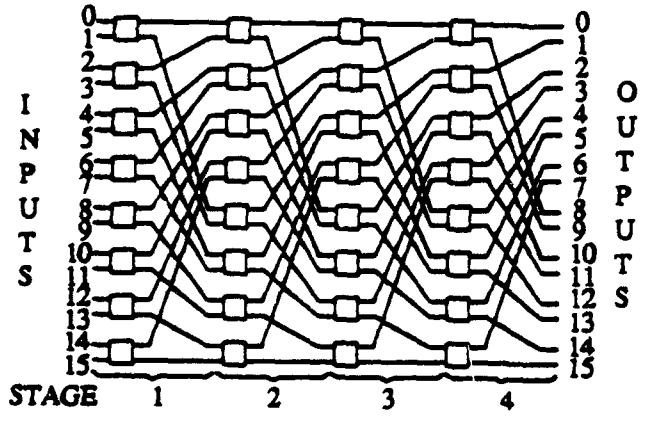


Figure II.6. The inverse omega network (i.e., the 4-stage inverse SE) with 16 inputs/outputs.

An *IN* having N inputs/outputs and k stages is denoted by both $IN_{N\times k}$ and $IN_{:k}$, where $k \geq 1$. The subnetwork that consists of the stages x through y of $IN_{1:k}$ is denoted by $IN_{x:y}$, where $1 \leq x < y \leq k$. If $x > y$, then $IN_{x:y}$ refers to a nil network, unless specified otherwise. IN_j, $1 \leq j \leq k$, refers to the jth stage of $IN_{1:k}$. The notation used for networks is different from that used for matrices because matrices are always denoted by single letters.

Without loss of generality, it is assumed that routing of a permutation through a network is done as described in this paragraph. Assuming that the stages of the network are labeled from left to right starting with 1, if the routing tag is $d_1 d_2 \cdots d_k$, then d_i is examined to set the SB at stage i as follows: if d_i equals zero then the output is sent to the upper output of the SB; otherwise, it is sent to the lower output. The ith entries of the routing tags of the two inputs entering a SB are also called the control bits of that SB. So, to set a SB properly to either straight or cross (or equivalently not to have any conflict in a SB), the control bits of a SB must constitute the set $\{0,1\}$. In some networks, the routing tag of an input equals its destination address, but this is not always the case.

In this paper the following convention is adopted to denote an IN: if the name of an IN has more than one word, then it is denoted by the upper case form of the first letters of those words; otherwise, it is denoted by the upper case form of its first and last letters. Also, if *XX* denotes an IN, then the *inverse XX* network may be denoted by XX^{-1}. The following definition applies this convention to the baseline, reverse baseline, shuffle-exchange, inverse shuffle-exchange, Benes and three-stage Clos networks of interest in this paper.

Definition II.3. (BE, RB, SE, SE^{-1}, BS, CS, composite IN): The symbols *BE*, *RB*, *SE*, SE^{-1}, *BS* and *CS* in this paper refer to the networks baseline, reverse baseline, shuffle-exchange, inverse shuffle-exchange, Benes and three-stage Clos network $v(2,2,N/2)$ [7,13], respectively. (If the number of inputs/outputs of three-stage Clos network $v(2,2,N/2)$ is equal to N, then each of the outside stages of three-stage Clos network in this paper contains $N/2$ (2×2) SBs and the middle stage consists of 2 boxes with $N/2$ inputs/outputs each). If an IN is a cascade of different INs, then it is called a *composite IN* and is denoted by the concatenation of symbols that represent the *INs* in the order they are cascaded.

As an example for a composite network, the notation $RB_{1:n}SE_{1:m}$, $m \geq 1$, denotes the network consisting of $RB_{1:n}$ followed by $SE_{1:m}$.

Linial and Tarsi [2] introduced the concept of balanced matrices to establish a relation between SE networks and their realizable permutations. The following definition is equivalent to the one given in [2].

Definition II.4. (Balanced matrix): Let $N=2^n$ and call a 0-1 matrix $A_{N\times k}$ *balanced* if either one of the following conditions is satisfied:

1. For $k \le n$, it consists of any k columns of the binary representation of a permutation on the set $\{0,1,\ldots,N-1\}$.
2. For $k>n$, every n consecutive columns form the binary representation of a permutation on the set $\{0,1,\ldots,N-1\}$.

As an example, two balanced matrices E and F are shown below. But notice that the matrix $[E\ F]$ is not balanced.

$$E=[e_1\ e_2\ e_3]=\begin{bmatrix}1&1&1\\1&0&0\\0&0&0\\0&1&0\\1&1&0\\0&0&1\\1&0&1\\0&1&1\end{bmatrix} \qquad F=[f_1\ f_2]=\begin{bmatrix}1&0\\1&0\\1&1\\1&1\\0&1\\0&1\\0&0\\0&0\end{bmatrix}$$

Definition II.5. (Pass, realize): A balanced matrix $A_{N\times k}$ (respectively, an *IN*) is said to *pass* a k-stage IN (respectively, a matrix $A_{N\times k}$) if no conflict occurs in the SBs of the IN when $A_{N\times k}(i)$ is used as the routing tag for the *ith* input of the IN. A network *IN realizes* a permutation represented by $B_{N\times n}$ if there is a network switch setting such that input i is sent to output $B(i)$ for all $i=0,1,\ldots,N-1$.

According to the last definition, in this paper, the phrases "an IN passes a balanced matrix" and "a balanced matrix passes an IN" are used alternatively. It is also assumed that only "one pass" is allowed through a network to realize a permutation. Therefore, the phrase "one pass" is omitted in the sequel. To emphasize the distinction between the meaning of the terms "pass" and "realize" as used in this paper, it is important to notice that matrix $A_{N\times k}$ in Definition II.5 does not necessarily correspond to the permutation realized by the network *IN*. Indeed, the ith row of $A_{N\times k}$ is the routing tag for input i and it is only when it equals the destination of input i that $A_{N\times k}$ is the permutation realized by *IN*; the cases in which this occurs will become clear in the remainder of the paper.

II.2. A Motivational Example

Consider permutations π_1 = (0 6)(1 2)(3 5 4)(7), π_2 = (0 2)(1 4 3 7)(5 6), and the reverse baseline network with 8 inputs/outputs, denoted by $RB_{8\times 3}$ and shown in Figure II.7a. A frame is illustrated in Figure II.7b. The binary representations of these permutations are given below:

$$\pi_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad \pi_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

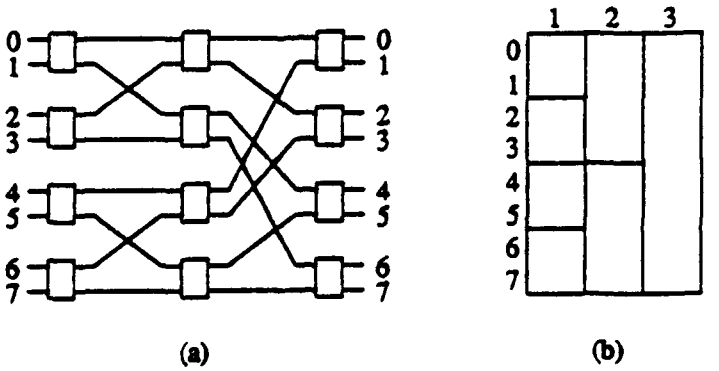


Figure II.7. (a) The reverse baseline network with 8 inputs/outputs.
(b) A *frame*.

When the *i*th row, $0 \le i \le 7$, of both π_1 and π_2 is used as the routing tag for the *i*th input of $RB_{8\times3}$, no conflict occurs in the switches and connections are established between the input i and the outputs $\Pi_1(i)$ and $\Pi_2(i)$, respectively. Therefore, $RB_{8\times3}$ realizes π_1 and π_2. Now, let us place the *i*th row of π_1 and π_2 into the *i*th row of the frame in Figure II.7b with 8 rows as shown in Figure II.8a and Figure II.8b, respectively.

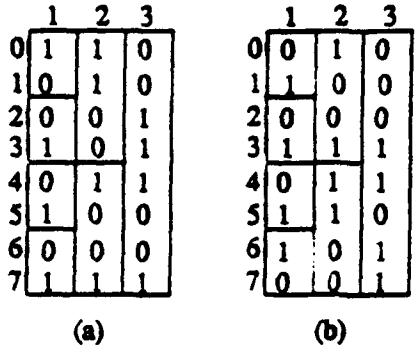


Figure II.8. (a) Frame with the binary representation of the permutation π_1.
(b) Frame with the binary representation of the permutation π_2.

The first k columns, $1 \le k \le 3$, of any of these two frames consists of 2^{3-k} rectangles of size $2^k \times k$. Note that the matrix enclosed by any rectangle of the frames is balanced (in fact, it represents a permutation on $\{0,1,\ldots,2^k\}$). It is shown in Section IV that, when the rows of any permutation realized by the reverse baseline network are placed into this type of frame, the matrix enclosed by each rectangle is balanced, and vice versa. Different frames are introduced in this paper and it is shown how they are useful to identify the permutations realized by some frequently used networks.

III. FRAMES AND FUNDAMENTAL CONCEPTS

This section introduces the concept of frames to characterize the permutations realized by a network. Different frames are derived from this concept and their graphical representations are presented. In addition, some related fundamental concepts used in the proofs of this paper are introduced. More extended discussion of these concepts appears in [28].

In order to facilitate the understanding of the concept of frame, the following definition is first introduced (a *k-tuple* V with the elements $v_1, v_2, \ldots, v_k$, denoted by $V = <v_1, v_2, \ldots, v_k>$, refers to an ordered collection of k elements).

Definition III.1. (Partition P_i, block, standard partition P_i^*, P^*): Let $X=\{0,1,\ldots,N-1\}$, $N=2^n$ and $i=1,2,\ldots,n$. A *partition* P_i of X is a tuple of 2^{n-i} disjoint ordered subsets of X, called *blocks*, each of which is a tuple with 2^i distinct elements. The partition $P_i^* = <<h, h+1, \ldots, h+2^i-1>$ such that $h \bmod 2^i = 0$ and $h = 0,1,\ldots,N-1>$ is a *standard partition* of X. The *n-tuple* $<P_i^*, i=1,2,\ldots,n>$ is denoted by P^*.

Example III.1. Let $N=8$. The following are the standard partitions:
$P_1^* = <<0,1>, <2,3>, <4,5>, <6,7>>$, $P_2^* = <<0,1,2,3>, <4,5,6,7>>$ and $P_3^* = <0,1,2,3,4,5,6,7>$. Also, $P^* = <P_1^*, P_2^*, P_3^*>$.

The notion of frame is defined next and an example (Example III.2) is given after the definition. Note that the frame of Figures II.7 and II.8 is characterized by the labeling of its columns, the labeling of its rows and how each column is partitioned. Therefore, the definition of frame is done in terms of two mappings (the column and row labeling) and a tuple of partitions (one for each column). The column labels determine the number and size of the blocks in each partition and the row labeling determines the elements in each block and their order. As precisely stated in the definition, column with label $\beta(i)$ corresponds to a partition with $2^{n-\beta(i)}$ blocks with $2^{\beta(i)}$ elements each and the mth element within the jth block corresponds to the label $\gamma(r)$ of row $r=2^{\beta(i)}(j-1)+m-1)$. After Example III.2, a convenient graphical representation for frames is introduced and its use is illustrated in Example III.3 for the frames described in Example III.2.

Definition III.2. (Frame): Let $1 \le k \le n$ and $1 \le i \le k$. A *frame* $F_{N\times k}$, $1 \le k \le n$, is a 3-tuple $<\beta,\gamma,P>$, where

- β is a mapping of the set $\{1,2,\ldots,k\}$ into $\{1,2,\ldots,n\}$,
- γ is a permutation on the set $\{0,1,\ldots,N-1\}$ and
- P is a tuple of partitions $<P_{\beta(1)},P_{\beta(2)},\ldots,P_{\beta(k)}>$ determined by β and γ as follows:
  $P_{\beta(i)} = <P_{\beta(i),1},P_{\beta(i),2},\ldots,P_{\beta(i),2^{n-\beta(i)}}>$ where
  $P_{\beta(i),j} = <u_{1,j},u_{2,j},\ldots,u_{2^{\beta(i)},j}>$ such that
  $u_{m,j} = \gamma(2^{\beta(i)}(j-1)+m-1)$ for $1 \le j \le 2^{n-\beta(i)}$ and $1 \le m \le 2^{\beta(i)}$.

Definition III.3. (a-frame, standard a-frame): Consider the 3-tuple $<\beta,\gamma,P>$ that defines a frame $F_{N\times k}$. If β is the identity permutation, then $F_{N\times k}$ is an *a-frame* denoted by $F^{a}_{N\times k}$. If β and γ are the identity permutations (which implies $P=P^{*}$), then $F_{N\times k}$ is the *standard a-frame* denoted by $F^{sa}_{N\times k}$.

By definition of standard *a*-type frame, column f^{sa}_{i}, $1 \le i \le n$, has 2^{n-i} blocks, each having 2^{i} rows. Unless otherwise stated, the number of the rows of $F^{sa}_{1:k}$, $k \ge 1$, is assumed to be N. Similar to the notation of matrices, to be able to refer to specific columns of a frame, the notation $F_{x:y}$ is used to denote the subframe that contains those columns of F whose indices are $x, x+1,\ldots,y$. Unless specifically stated, the number of rows of $F_{x:y}$ is assumed to be N.

Example III.2. The following are examples of frames for $N=8$ and $k=3$.
(*a*) $F_{8\times3}=<\beta,\gamma,P>$ where $\beta=(1\,2)(3)$, γ is the identity permutation and $P=<P_2,P_1,P_3>$ such that $P_1=P^{*}_1$, $P_2=P^{*}_2$ and $P_3=P^{*}_3$.
(*b*) $F^{a}_{8\times3}=<\beta,\gamma,P>$ where $\beta=$ identity permutation, $\gamma=(0)(1\,2)(3)(4)(5)(6)(7)$, $P=<P_1,P_2,P_3>$, $P_1=<<0,2>, <1,3>, <4,5>, <6,7>>$, $P_2=<<0,2,1,3>, <4,5,6,7>>$ and $P_3=<0,2,1,3,4,5,6,7>$.

(*c*) $F^{a}_{8\times3}=<\beta,\gamma,P>$ where β = identity permutation, $\gamma=(0)(1\ 3\ 6\ 4)(2\ 5)(7)$, $P=<P_1,P_2,P_3>$, $P_1=<<0,3>, <5,6>, <1,2>, <4,7>>$, $P_2=<<0,3,5,6>, <1,2,4,7>>$ and $P_3=<0,3,5,6,1,2,4,7>$.

(*d*) $F_{8\times3}=<\beta,\gamma,P>$ where $\beta=\begin{bmatrix}1&2&3\\2&2&3\end{bmatrix}$, γ is the identity permutation, $P=<P_2,P_2,P_3>$, $P_2=P_2^*$ and $P_3=P_3^*$.

Definition III.4. (Graphical representation of a frame, rectangle of a frame): The *graphical representation of a frame* $F_{N\times k}=<\beta,\gamma,P>$ consists of k columns labeled f_i, $i=1,\ldots,k$, from left to right and N rows labeled $\gamma(j)$, $j=0,1,\ldots,N-1$ starting at the top. The column f_i corresponds to the partition $P_{\beta(i)}$, that is, f_i consists of $2^{n-\beta(i)}$ blocks of $2^{\beta(i)}$ entries each. In the graphical representation of a frame, any polygon with four sides and four right angles is a *rectangle of the frame*.

Example III.3. Figures III.1a, III.1b, III.1c and III.1d show the graphical representation of the frames described in the part (*a*), (*b*), (*c*) and (*d*) of Example III.2, respectively. Figure III.1e shows the graphical representation of the standard *a*-frame $F^{sa}_{8\times3}$. The labels of the partitions below each column are implicit by the sizes of the rectangles in the column and can be omitted.

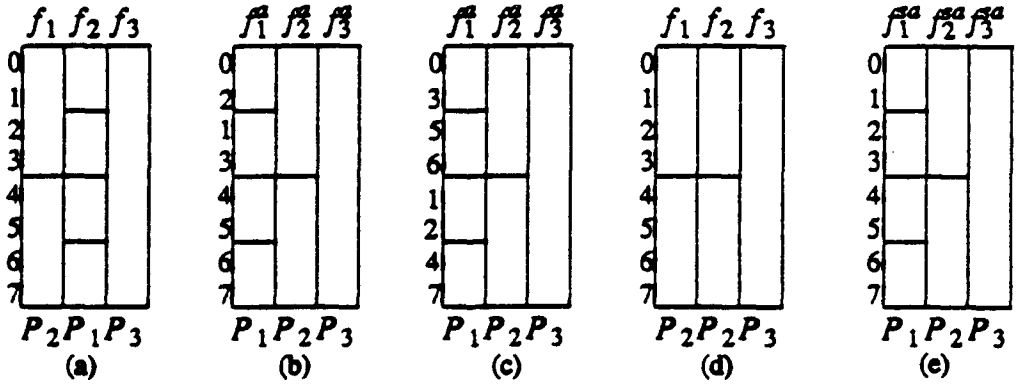


Figure III.1. (*a*), (*b*), (*c*) and (*d*) are the graphical representations of the frames described in the part (*a*), (*b*), (*c*) and (*d*) of Example III.2, respectively. (*e*) Graphical representation of the standard a-frame $F^{sa}_{8\times3}$.

Definition III.5. (Fit): Let $k\geq1$, $0\leq i\leq N-1$ and $1\leq j\leq k$. Consider a balanced matrix $A_{N\times k}$ and a frame $F_{N\times k}$. The matrix A *fits* $F_{N\times k}$ if and only if, after placing a_{ij} in the ith row and jth column of $F_{N\times k}$, every rectangle of $F_{N\times k}$ contains a balanced matrix.

Example III.4. The matrix E, shown just after Definition II.4, fits all the frames shown in Figures III.1 except $F^{sa}_{8:3}$ shown in Figure III.1e because, for example, the submatrix in the top leftmost rectangle (the 2-tuple $P_{1,1}$) is not balanced.

Note that the value of k in Definition III.5 does not have to equal n. It will become clear that frames of any number of columns can be used to characterize permutations (which are represented by balanced matrices of n columns).

In addition to a-frames, other two types of frames are of use in this paper. One is called universal frame and, as suggested by its name, any balanced matrix fits it. The other type of frame is a concatenation of frames and is useful in characterizing the permutations realized by, for example, composite networks.

Definition III.6. (Universal frame $F^{*}_{1:k}$): The *universal frame* $F^{*}_{1:k}$, $k \geq 1$, is such that, for $i=1,2,\ldots,k$, $\beta(i)=n$, γ is the identity permutation and $P_i = P^{*}_{n}$. The universal frame $F^{*}_{1:k}$ is illustrated in Figure III.2.

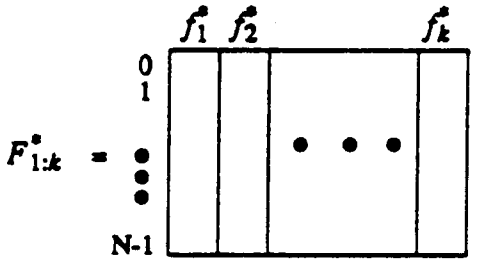


Figure III.2. The universal frame $F^{*}_{1:k}$.

Definition III.7. ($F^{sa}_{1:n}F^{*}_{1:m}$): The notation $F^{sa}_{1:n}F^{*}_{1:m}$, $m \geq 1$, represents the frame obtained by concatenating $F^{sa}_{1:n}$ and $F^{*}_{1:m}$ as shown in Figure III.3.

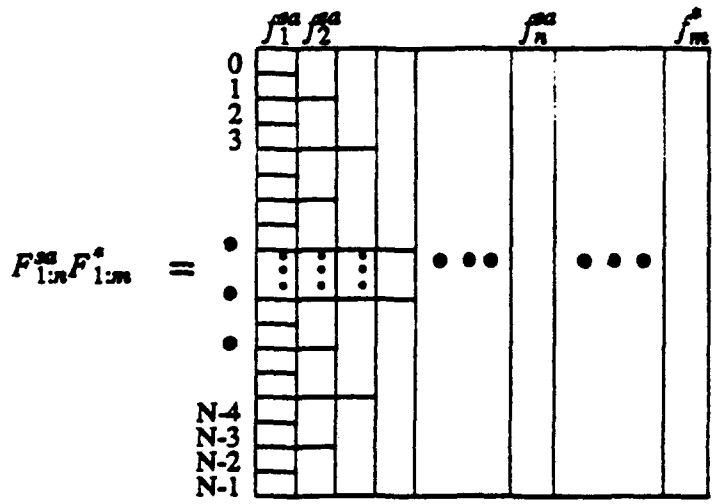


Figure III.3. The frame $F_{1:n}^{aa}F_{1:m}^{a}$ which is obtained by concatenating $F_{1:n}^{aa}$ and $F_{1:m}^{a}$.

The following definition states precisely what means to establish a correspondence between a frame and a network.

Definition III.8. (Correspondence between frames and networks): A frame (respectively, an IN) is said to *correspond* to an IN (respectively, a frame) if a balanced matrix fits the frame if and only if it passes the network.

When a frame corresponds to a network it suffices to check if a matrix fits the frame in order to determine whether the network passes the matrix. This does not mean that, when the matrix represents a permutation, the network realizes the permutation. Instead, it means that, when the rows of the matrix are used as routing tags, no conflicts occur in the network.

The complexity of checking that a matrix fits a frame is discussed next. First, the complexity of testing if a rectangle contains a permutation matrix is considered. Next, the complexity of checking all rectangles of the same size is discussed and, finally, the complexity of checking all rectangles (i.e., the entire frame) is derived. Note that it suffices to consider only those rectangles whose number of columns equals the logarithm of the number of rows.[1] To check whether a given rectangle with x rows and $\log x$ columns contains a balanced matrix, it suffices to verify that the rows of the matrix are distinct. This can be done by building a binary search tree starting with the root which corresponds to the first row of the matrix; each row is then added as a leaf to the tree as long as it is distinct from all previously inserted rows and so that it satisfies the binary-search-tree property [29]. According to this property, if $v(p)$ is the value of the row that corresponds to node p, then $v(y)<v(p)$ for any node y in the left

[1] All logarithms are in base 2 unless stated otherwise.

subtree of p and $v(z)>v(p)$ for any node z in the right subtree of p. In the worst case, this procedure takes $O(x^2)$ steps and has average complexity of $O(x\log x)$ [29]. If several rectangles of the same size exist in a frame, then the total number of rows contained in all the rectangles with the same columns is N. The same procedure can be used for each rectangle and the total worst case and average complexities will be $O(N^2)$ and $O(N\log N)$, respectively. Because there are at most k different types of rectangles in a frame with k columns, the total worst case and average complexities are $O(kN^2)$ and $O(kN\log N)$, respectively. These bounds apply to any frame, but it is possible to do better with particular frames. For example, for a-frames the worst-case complexity becomes $O(N^2+2(N/2)^2+\cdots+(N/2)2^2)=O(N^2)$.

IV. BASELINE-TYPE NETWORKS

Equivalence relations among INs have been extensively studied in the literature using different tools such as graph theory, group theory, and Boolean algebra [6,11,27,26]. Networks can be modeled by directed graphs where vertices and edges represent switches and links, respectively. Two INs are *functionally equivalent* if they realize the same set of permutations while two INs are *topologically equivalent* if their topologies (i.e., directed graphs) are isomorphic. Wu and Feng [11] have shown the topological equivalence of a class of MINs, which include data manipulator [14], omega [1], flip [15], SW-banyan (s=f=2) [16], and indirect binary n-cube [17], baseline and reverse baseline [11]. From [18], "the notion of functional equivalence is more practical than that of topological equivalence because it provides an equivalence basis among networks at their inputs, and thus it does not call for any modification in their internal switching structure". Given a network in a class of isomorphic INs, it is possible to rename its inputs and/or outputs so that this network can directly simulate any network in the class [11]. In this section, all the matrices that pass those networks that are topologically equivalent to the k-stage baseline, $1 \le k \le n$, are identified by a-frames that may differ only in how their rows are labeled. First, the permutations realized by the k-stage reverse baseline are identified. Then, this result is extended to the other networks. These results also show how the addition of a stage to the right of these networks changes the type of their realizable permutations. An algorithm is provided to find whether a network is topologically equivalent to the reverse baseline network, its corresponding frame and how to relabel inputs and outputs to achieve functional equivalence. Omitted proofs are provided in the Appendix.

IV.1. Correspondence between $F^{sa}_{1:k}$ and $RB_{1:k}$

Because $RB_{1:n}$ is functionally and topologically equivalent to $BE_{1:n}$ [11], any permutation that is realized by $RB_{1:n}$ is also realized by $BE_{1:n}$, and vice versa. However, this is not true for $RB_{1:k}$ and $BE_{1:k}$, $1 \le k \le n-1$, because they are not functionally equivalent (they are only topologically equivalent). But, the set of balanced matrices that pass $RB_{1:k}$ is the same as the set of balanced matrices that pass $BE_{1:k}$ as explained next. The network $RB_{1:k}$ can be obtained by repositioning the SBs of the second stage through the last stage of $BE_{1:k}$ and reordering its outputs. It follows that any pair of routing tags that enter a SB at the kth stage of $BE_{1:k}$ also enter a SB at the kth stage of $RB_{1:k}$, and vice versa. So, if the routing tags used in $BE_{1:k}$ do not create any conflict, then they also do not have any conflict in the SBs of $RB_{1:k}$, and vice versa. Therefore, a balanced matrix $D_{1:k}$ passes $RB_{1:k}$ if and only if $D_{1:k}$ passes $BE_{1:k}$.

The following theorem shows that there exists a very close relation between $RB_{1:k}$ and $F^{sa}_{1:k}$, $1 \le k \le n$, so that the matrices that pass the network can be identified by $F^{sa}_{1:k}$. It shows that the ith input of $RB_{1:k}$ is sent without conflicts to the output whose value equals $\left(\left\lfloor i/2^k \right\rfloor \times 2^k\right)$ plus the value of $D_{1:k}(i)$ when the ith row of a matrix $D_{1:k}$ that fits $F^{sa}_{1:k}$ is used as the routing tag for the ith input of $RB_{1:k}$.

Theorem IV.1 A matrix $D_{1:k} = [d_1\ d_2\ \cdots\ d_k]$ fits $F^{sa}_{1:k}$ if and only if $D_{1:k}$ passes $RB_{1:k}$, $1 \le k \le n$. Moreover, $RB_{1:k}$ sends its ith input to its jth output, where j is equal to the sum of $\left(\left\lfloor i/2^k \right\rfloor \times 2^k\right)$ and the value of $D_{1:k}(i)$.

Basic idea of proof (complete proof appears in Appendix):

($\rightarrow$) $D_{1:k}$ fits $F^{sa}_{1:k} \rightarrow D_{1:k}$ passes $RB_{1:k}$.

Induction on k is used. For $k=1$, each rectangle of $F^{sa}_{1:k}$ has a 0 and a 1. These correspond to the control bits of a switch in $RB_{1:k}$ and, thus, no conflict occurs. For $k>1$, assuming the theorem holds for $k-1$, it is also shown that each switch in the kth stage "has" control bits 0 and 1 and, therefore, no conflicts occur. These control bits must appear as the kth bits at the end of identical $(k-1)$-bit rows of subframes $F^{\alpha_1}_{2^{k-1}\times k-1}$ and $F^{\alpha_2}_{2^{k-1}\times k-1}$ of $F^{sa}_{1:k}$ so that $D_{1:k}$ fits $F^{sa}_{1:k}$. Each subframe corresponds to a subnetwork of $RB_{1:k}$ which is also a reverse baseline network $RB_{2^{k-1}\times k-1}$.

($\leftarrow$) $D_{1:k}$ passes $RB_{1:k} \rightarrow D_{1:k}$ fits $F^{sa}_{1:k}$.

Induction on k is used. For $k=1$, if d_1 passes $RB_{1:}$, then each rectangle of $F^{sa}_{1:k}$ contains a 0 and a 1 and d_1 fits f^{sa}_1. For $k>1$, assuming the theorem holds for $k-1$, it is shown that for the outputs of two subnetworks $RB^{\alpha_1}_{2^{k-1}\times k-1}$ and $RB^{\alpha_2}_{2^{k-1}\times k-1}$ to cause no conflict in any switch of the kth stage it must be the case that a 0 and a 1 are added to the $k-1$ entries of identical rows of the frames that correspond to the two subnetworks. This implies that $D_{1:k}$ fits $F^{sa}_{1:k}$. The value of j follows from the topology of $RB_{1:k}$ and how switches are set by control bits. □

Corollary IV.1. A network with k stages and N inputs/outputs is topologically equivalent to the k-stage reverse baseline, $RB_{1:k}$, if and only if it corresponds to an a-type frame $F^a_{1:k}$, where $1 \le k \le n$.

IV.2. Permutations Realized by Baseline-Type Networks

In this section, a-type frames are used to characterize all the permutations realized by any network that is topologically equivalent to the baseline network. An algorithm, called FRAME_IN, is introduced to determine the a-type frame that corresponds to a given network. It is also shown how to construct a network to realize all the permutations that fit an a-type frame.

Let $F^a_{1:k}(\alpha^{-1})$ denote a particular a-type frame where $\gamma=\alpha^{-1}$, i.e., whose row labels form the vector α^{-1}. Let Π denote a network with k stages which is the same as $RB_{1:k}$ except that the label of its ith input equals the ith entry of α^{-1}. By Corollary IV.1, a balanced matrix $D_{1:k}$ fits $F^a_{1:k}(\alpha^{-1})$ if and only if $D_{1:k}$ passes Π. If $k=n$, any of these balanced matrices represents a permutation, so that Π is a network that realizes all the permutations characterized by $F^a_{1:k}(\alpha^{-1})$. If $k<n$, then the relation between a $D_{1:k}$ that fits $F^a_{1:k}(\alpha^{-1})$ and a permutation that passes Π is first determined. By applying this relation to every balanced matrix that fits $F^a_{1:k}(\alpha^{-1})$, all the permutations realized by Π are determined. Theorem IV.3 determines the relation between a balanced matrix that fits $F^{sa}_{1:k}$ and the permutation realized by $RB_{1:k}$ when this balanced matrix passes the network. Corollary IV.3 generalizes this result to the class of baseline-type networks.

Theorem IV.3. A matrix $D_{1:k}$, $1 \le k \le n$, fits $F^{sa}_{1:k}$ if and only if $RB_{1:k}$ realizes the permutation represented by $[I_{1:n-k}\, D_{1:k}]$.

Proof. ($\rightarrow$) Let $D_{1:k}$ fit $F^{sa}_{1:k}$. It is shown that $RB_{1:k}$ realizes the permutation represented by $[I_{1:n-k}\, D_{1:k}]$.

Theorem IV.1 states that $RB_{1:k}$ sends its ith input, $0 \le i \le N-1$, to its jth output, where j is equal to the sum of $\left(\left\lfloor i/2^k \right\rfloor \times 2^k\right)$ and the value of $D_{1:k}(i)$. Due to the fact that $[\left(\left\lfloor i/2^k \right\rfloor \times 2^k\right)+D_{1:k}(i)]$ equals the ith row of $[I_{1:n-k}\, D_{1:k}]$, $RB_{1:k}$ realizes the permutation represented by $[I_{1:n-k}\, D_{1:k}]$.

($\leftarrow$) Assume that $RB_{1:k}$ realizes the permutation represented by $[I_{1:n-k}\, D_{1:k}]$. It is shown that $D_{1:k}$ fits $F^{sa}_{1:k}$.

Because $RB_{1:k}$ realizes the permutation represented by $[I_{1:n-k}\, D_{1:k}]$, it sends its ith input to the output whose value equals the sum of $\left(\left\lfloor i/2^k \right\rfloor \times 2^k\right)$ and the value of $D_{1:k}(i)$, $D_{1:k}$ passes $RB_{1:k}$. It follows from Theorem IV.1 that $D_{1:k}$ fits $F^{sa}_{1:k}$. □

Corollary IV.3. Consider a k-stage, $1 \le k \le n$, network Π which is topologically equivalent to $RB_{1:k}$. The network Π is functionally and topologically equivalent to a network $IP_{in}RB_{1:k}IP_{out}$, where IP_{in} and IP_{out} are interconnection patterns that realize permutations α_{in} and α_{out}, respectively. Also, let $F^a_{1:k}(\alpha_{in}^{-1})$ denote an a-type k-column frame whose ith row label equals $\alpha_{in}^{-1}(i)$ for $i=0,1,\ldots,N-1$. A matrix $D_{1:k}$ fits $F^a_{1:k}(\alpha_{in}^{-1})$ if and only if Π realizes the permutation $\alpha_{in}.\mu.\alpha_{out}$, where μ is the permutation represented by the balanced matrix $[I_{1:n-k}\ D^*_{1:k}]$ and $D^*_{1:k}(i) = D_{1:k}(\alpha_{in}^{-1}(i))$.

Corollary IV.3 implies that the network $IP_{in}RB_{1:k}$ corresponds to the frame $F^a_{1:k}(\alpha_{in}^{-1})$, where IP_{in} realizes the permutation α_{in}. Hence, for a given $F^a_{1:k}$, a corresponding network can be constructed easily. The following example shows the construction of a network that realizes a set of permutations which includes two given permutations.

Example IV.1. Let N=16, k=2, $0 \le i \le N-1$. Assume that α_{in} and α_{out} denote the permutations realized by the interconnection patterns IP_{in} and IP_{out}. Given two permutations a = (0 9 8 5 1 2 12 10 14 6 3 7 11 13)(4)(15) and b = (0 7)(1)(2 3 9 13 11 8 4 5)(6 12)(10 15 14), it is shown how to construct a network $IP_{in}RB_{1:2}IP_{out}$ that realizes a set of permutations including a and b. Let A and B refer to the binary representations of a and b, respectively. By Theorem IV.3, any permutation that passes $RB_{1:2}$ must be represented by a balanced matrix whose first (leftmost) two columns form $I_{1:2}$ (recall that k=2 and n=4 in this example). If there was only one given permutation, then the balanced matrix representing the permutation could be converted by IP_{in} to a balanced matrix whose first two columns form $I_{1:2}$ because IP_{in} can be chosen so as to permute the rows in any given way. However, if more than one permutation are given, and the first two columns of their binary representations do not form the same matrix, then IP_{out} is needed to convert the binary representations of these permutations into balanced matrices whose first two columns form the same matrix. So, the matrices A and B are first converted by IP_{out} to $\dot{A}$ and $\dot{B}$ such that $\dot{A}_{1:2} = \dot{B}_{1:2}$. Specifically, α_{out} converts A and B to $\dot{A}$ and $\dot{B}$, respectively such that $\dot{A}(i) = \alpha_{out}^{-1}(A(i))$ and $\dot{B}(i) = \alpha_{out}^{-1}(B(i))$. Then, $\dot{A}$ and $\dot{B}$ are converted by α_{in} to $\ddot{A}$ and $\ddot{B}$, respectively such that the first two columns of each of these matrices form $I_{1:2}$. Specifically, $\ddot{A}(i) = \dot{A}(\alpha_{in}^{-1}(i))$ and $\ddot{B}(i) = \dot{B}(\alpha_{in}^{-1}(i))$. For instance, α_{out} = (0 13 12)(1 5 7 2 4)(3 8 6 9 14)(10 15 11) converts a and b to $\dot{a}$ = (0 6 14 8 1 7 15 10 9 3 5 4 2 13 12 11) and $\dot{b}$ = (0 5 7 12 8 2 14 11 3 6 13 15 9 0 5 7), respectively. Similarly, α_{in} = (0 5 4 1 7 15 9 3 6 13 12 11)(2 14 8)(10) converts $\dot{a}$ and $\dot{b}$ into $\ddot{a}$ = (0)(1 2)(3)(4)(5 6)(7)(8)(9 10)(11)(12)(13 14)(15) and $\ddot{b}$ = (0 3)(1)(2)(4 7)(5)(6)(8 11)(9)(10)(12 15)(13)(14), respectively. The binary representations of a, $\dot{a}$, $\ddot{a}$, b, $\dot{b}$ and $\ddot{b}$ are shown below. The network that realizes the

permutations a and b is shown in Figure IV.1.

A:

0	1001
1	0010
2	1100
3	0111
4	0100
5	0001
6	0011
7	1011
8	0101
9	1000
10	1110
11	1101
12	1010
13	0000
14	0110
15	1111

B:

0	0111
1	0001
2	0011
3	1001
4	0101
5	0010
6	1100
7	0000
8	0100
9	1101
10	1111
11	1000
12	0110
13	1011
14	1010
15	1110

$\dot{A}$:

0	0110
1	0111
2	1101
3	0101
4	0010
5	0100
6	1110
7	1111
8	0001
9	0011
10	1001
11	0000
12	1011
13	1100
14	1000
15	1010

$\dot{B}$:

0	0101
1	0100
2	1110
3	0110
4	0001
5	0111
6	1101
7	1100
8	0010
9	0000
10	1010
11	0011
12	1000
13	1111
14	1011
15	1001

$\ddot{A}$:

0	0000
1	0010
2	0001
3	0011
4	0100
5	0110
6	0101
7	0111
8	1000
9	1010
10	1001
11	1011
12	1100
13	1110
14	1101
15	1111

$\ddot{B}$:

0	0011
1	0001
2	0010
3	0000
4	0111
5	0101
6	0110
7	0100
8	1011
9	1001
10	1010
11	1000
12	1111
13	1101
14	1110
15	1100

Because a 2×2 switch has two possible settings (cross and straight), the number of balanced matrices that pass a k-stage baseline-type network with N inputs equals $2^{kN/2}$. By Corollary IV.1, for any given k-column a-type frame, there exists a corresponding baseline-type network. Therefore, exactly $2^{kN/2}$ balanced matrices fit any $F^a_{1:2}$. For k=2, 2^N balanced matrices pass a baseline-type network. Let $D^r_{1:2}$, $1 \le r \le 2^N$, denote one of the 2^N balanced matrices that fit $F^a_{1:2}(\alpha_{in}^{-1})$. Also, assume that $D^{*r}_{1:2}$ is obtained from $D^r_{1:2}$ such that $D^{*r}_{1:2}(i) = D^r_{1:2}(\alpha_{in}^{-1}(i))$. Let μ_r denote the permutation represented by $[I_{1:2}\ D^{*r}_{1:2}]$. So, the network shown in Figure IV.1 realizes any of those permutations that result from $\alpha_{in}.\mu_r.\alpha_{out}$. The ith row of $D^{*r}_{1:2}$ is used as the routing tag for the ith input of $RB_{1:2}$ in $IP_{in}RB_{1:2}IP_{out}$. As an example, let r=1 and consider the balanced matrix $D^1_{1:2}$, shown in Figure IV.2a, that fits $F^a_{1:2}(\alpha_{in}^{-1})$. The matrix $D^{*1}_{1:2}$ that is obtained from $D^1_{1:2}$, and the matrix $[I_{1:2}\ D^{*1}_{1:2}]$ are also shown in Figure IV.2. When the ith row of $D^{*1}_{1:2}$ is used as the routing tag for the ith input of $RB_{1:2}$, $RB_{1:2}$ realizes the permutation μ_1 = (0)(1 3 2)(4 5 6)(7)(8 10 11)(9)(12 15 14 13) which is represented by $[I_{1:2}\ D^{*1}_{1:2}]$. On the other hand, the network $IP_{in}RB_{1:2}IP_{out}$ realizes the permutation (0 9 4 8 5 7 3 1 2 12 6)(10)(11 13)(14 15) which results from $\alpha_{in}.\mu_1.\alpha_{out}$. End of example.

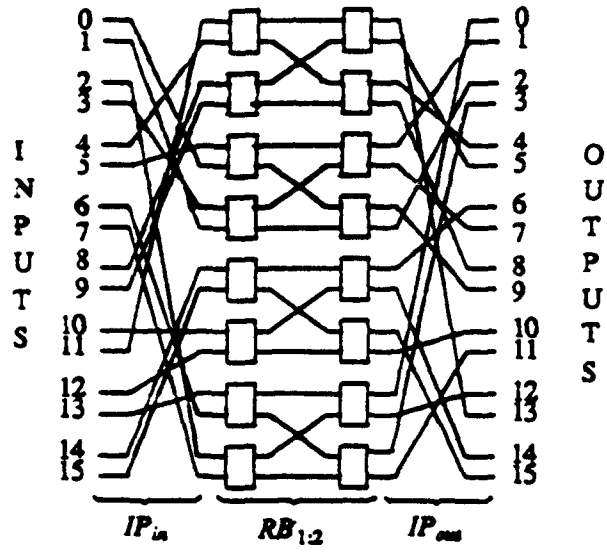


Figure IV.1. The network $IP_{in}RB_{1:2}IP_{out}$ of Example IV.1.

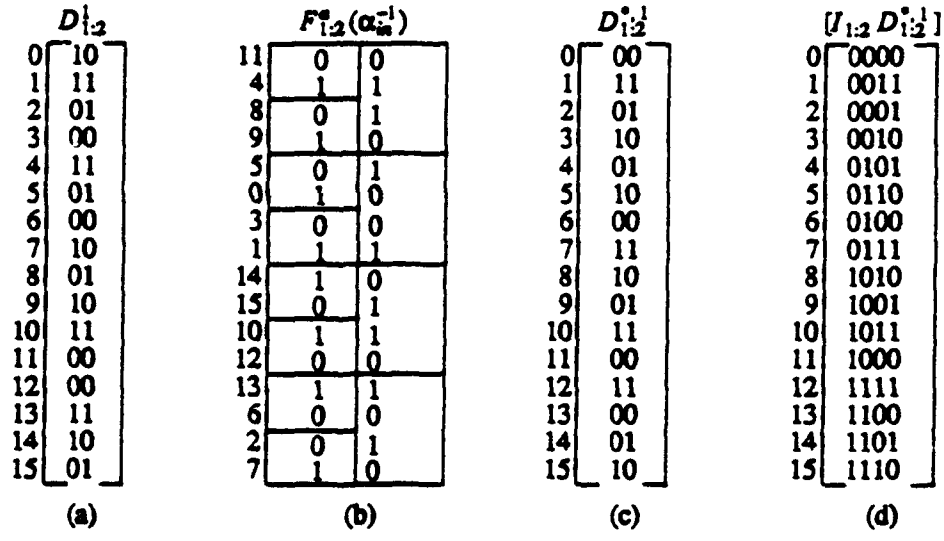


Figure IV.2. (a) A balanced matrix $D^1_{1:2}$ which fits $F^e_{1:2}(\alpha^{-1}_{in})$. (b) $F^e_{1:2}(\alpha^{-1}_{in})$ with $D^1_{1:2}$. (c) $D^{e,1}_{1:2}$ whose ith row equals $D^1_{1:2}(\alpha^{-1}_{in}(i))$. (d) $[I_{1:2}\ D^{e,1}_{1:2}]$.

In the rest of this section, some preliminary results used in the Algorithm FRAME_IN are first presented, then the algorithm is introduced.

Lemma IV.1. Let r denote the reverse permutation represented by the reverse permutation matrix $R_{N\times n}$ described in Definition II.1. The reverse baseline network $RB_{1:n}$ realizes r when all the switches are set straight.

Proof. The permutation realized by $RB_{1:n}$ when all the switches are set straight is determined by the interconnection patterns $IP_{in}, IP_1, \ldots, IP_{n-1}, IP_{out}$. Because $IP_{in} = IP_{out}$ = identity pattern, the permutation is given by $\alpha_1.\alpha_2 \cdots \alpha_{n-1}$ where α_i is the permutation realized by IP_i. Permutation α_i is such that $\alpha_i(x)$ rotates left the rightmost $i+1$ bits of x by one position because IP_i is a pile of 2^{n-i-1} shuffle-exchange patterns each with 2^{i+1} links. Applying this operation for all i starting with the initial matrix $I_{N\times n}$ yields the reverse permutation matrix $R_{N\times n} = [i_n\ i_{n-1} \cdots i_1]$. □

Because the reverse baseline network can be converted to the baseline network by repositioning the switches of the middle stages only, Lemma IV.1 is also valid for the baseline network. If there exists a unique path between any input and any output of a network, then the network satisfies the *Banyan property* [6,26]. Bermond et. al [26] present a set of properties to determine whether a network is topologically equivalent to baseline network. Their main result is formally restated below.

Theorem IV.2. [26] Let G be a directed graph representing a network with n stages and N inputs/outputs which satisfies the Banyan property. This network is topologically equivalent to the baseline network if and only if both the first j stages and the last j stages of G contain 2^{n-j} connected components for each j, $1 \le j \le n$.

This result is used next as the basis for Algorithm FRAME_IN. The description of the algorithm is followed by a proof of its correctness and analysis of its complexity.

Algorithm FRAME_IN

Input: A network GN with 2×2 switches, n stages and 2^n inputs/outputs.

Output: An a-type frame that corresponds to GN if GN is topologically equivalent to the baseline network; the permutations α_{in} and α_{out} realized by the interconnection patterns IP_{in} and IP_{out}, respectively, such that the network $IP_{in}GN_{1:n}IP_{out}$ is functionally equivalent to $RB_{N\times n}$.

Step 1. Let G denote a graph with n "stages" that is obtained by representing the switches and links of the given network by vertices and edges that are directed from left to right, respectively.

Step 2. Using a breadth-first search algorithm check whether there exists a unique path between any input vertex and any output

vertex of G. If so, go to next step. If not, go to Step 9.

Step 3. Let j and p be integer variables initialized to 0.

Step 4. Increment j by 1. If $j>n$, then go to next step; otherwise, using a depth-search algorithm, check whether the last j stages of the G contain 2^{n-j} connected components. If so, go to Step 4. If not, go to Step 9.

Step 5. Increment p by 1. If $p>n$, then go to Step 7; otherwise, using a depth-search algorithm, check whether the first p stages of G contains 2^{n-p} connected components. If so, go to next step. If not, go to Step 9.

Step 6. If $p=1$, let V_r^1 denote a vector of the input labels of a distinct connected component (a 2×2 switch) for each r, $(1 \le r \le 2^{n-1})$, and then go to Step 5; otherwise, do: let V_r^p, $1 \le r \le 2^{n-p}$, denote a vector that is formed by merging two vectors V_s^{p-1} and V_t^{p-1} for $1 \le s, t \le 2^{n-p+1}$ and $s \ne t$ such that the set of entries of V_r^p equals the set of input labels of a distinct connected component determined in Step 5. Go to Step 5.

Step 7. Let $\gamma(i) = V_1^n(i)$ for $i = 1,2,...,N-1$ (note that V_1^n is obtained in Step 6). Write "The a-type frame $F_{1:n}^a$ whose ith row label equals $\gamma(i)$ corresponds to the GN".

Step 8. Let σ denote the permutation realized by the given network $GN_{1:n}$ when all the switches are set straight. The permutation realized by IP_{in} is $\alpha_{in} = \gamma^{-1}$. The permutation realized by IP_{out} is $\alpha_{out} = \sigma^{-1}.\alpha_{in}^{-1}.r$, where r is the reverse permutation represented by the reverse permutation matrix $R_{N\times n}$ (see Definition II.1). Stop.

Step 9. Write "The given network is not topologically equivalent to baseline network and no corresponding a-type frame exists". Stop.

In Steps 2 through 6, Algorithm FRAME_IN checks whether the given network satisfies the set of properties described in Theorem IV.2. Specifically, Step 2 checks the Banyan property, while Steps 3 through 6 check whether both the first j stages and the last j stages of the network graph contain 2^{n-j} connected components, for each j. So, if Algorithm FRAME_IN fails in any of these steps, then it follows from Theorem IV.2 that the given network is not topologically equivalent to baseline network and, by Corollary IV.1, has no corresponding a-type frame.

It is now shown that the given network corresponds to the a-type frame determined in Step 7, that is, any balanced matrix that fits the a-type frame determined in

Step 7 passes the given network, and vice versa. Theorem IV.1 proves that, for $1 \le k \le n$, the frame $F^{sa}_{1:k}$ corresponds to $RB_{1:k}$, that is, a balanced matrix $D_{1:k}$ fits $F^{sa}_{1:k}$ if and only if $D_{1:k}$ passes $RB_{1:k}$. Note that $RB_{1:j}$ is a pile of 2^{n-j} $RB_{2i,j}$s. Recall that the only difference between the standard a-frame $F^{sa}_{1:k}$ and an a-type frame $F^{a}_{1:k}$ is the order of their row labels. Because Step 7 assigns $\gamma(i)$ to the ith row label of $F^{a}_{1:n}$, this frame corresponds to the given network. Step 8 first assumes that the permutation realized by the given network equals σ when all the switches are set straight. Then, Step 8 states that the interconnection pattern IP_{in} realizes the permutation $\alpha_{in} = \gamma^{-1}$. Relabeling the ith input of the given network by $\gamma(i)$ is equivalent to adding the interconnection pattern IP_{in} to the left of the given network. Thus, any balanced matrix that fits the a-type frame obtained in Step 7 passes the network $IP_{in}GN_{1:n}$, and vice versa. Algorithm FRAME_IN also adds an interconnection pattern IP_{out} that realizes a permutation called α_{out} to the right of the given network such that the network $IP_{in}GN_{1:n}IP_{out}$ realizes the permutation r when all the switches are set straight. By Lemma IV.1, the reverse baseline (baseline) realizes the permutation r when all the switches are set straight. Therefore, the network $IP_{in}GN_{1:n}IP_{out}$ is functionally and topologically equivalent to the reverse baseline and baseline networks. This completes the proof of correctness of the algorithm.

The graph of Algorithm FRAME_IN can have at most $O(N\log N)$ vertices because each vertex represents a switch. Algorithm FRAME_IN uses a breadth-first search to check whether the given network holds the Banyan property. A depth-first search is used to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components [29]. If V and E are the sets of vertices and edges, respectively, the running time of both a breadth-first search and a depth-first search is $\Theta(V+E)$. This implies that, for each value of j, Algorithm FRAME_IN takes $\Theta(N\log N)$ time. Because there are $2\log N$ iterations, the running time of Algorithm FRAME_IN is $\Theta(N \log^2 N)$.

Algorithm FRAME_IN yields a frame that corresponds to the given network. This means that any matrix that fits the frame also passes the network and vice versa. However, this does not necessarily mean that the permutation represented by the matrix is realized by the network. When a balanced matrix $D_{N\times n}$ fits an a-frame corresponding to a baseline-type network, the network realizes the permutation $d \,.\, \alpha_{out}$, where d is the permutation represented by $D_{N\times n}$ and α_{out} is the permutation realized by IP_{out} determined in Step 8 of Algorithm FRAME_IN. In other words, given a network that is topologically equivalent to the reverse baseline, relabeling its inputs and outputs by α_{in}^{-1} and α_{out}, respectively, results in a new network that is functionally equivalent to the reverse baseline.

As an example, for N=16, Algorithm FRAME_IN can be used to characterize the permutations of the following baseline-type networks: generalized cube, omega,

indirect binary n-cube, banyan (S=F=2), inverse omega, modified data manipulator, flip. The topological equivalence among these networks and baseline and reverse baseline networks is well known and previously studied in [6,11,18,26]. From Corollary IV.1, each of these networks corresponds to an a-frame. Algorithm FRAME_IN yields the row labeling γ and α_{out} for each of these networks and frames as follows: $\gamma = \alpha_{out}$ = identity permutation for the reverse baseline and baseline networks, γ = the reverse permutation = (0)(1 8)(2 4)(3 12)(5 10)(6)(7 14)(9)(11 13)(15) and α_{out} = identity permutation for the omega and generalized cube, γ = identity permutation and α_{out} = (0)(1)(2 8)(3 9)(4)(5)(6 12)(7 13)(10)(11)(14)(15) for the indirect binary cube, banyan, inverse omega, and flip networks, γ = (0)(1)(2 8)(3 9)(4)(5)(6 12)(7 13)(10)(11)(14)(15) and α_{out} = identity permutation for the modified data manipulator network.

V. NETWORKS $RB_{1:n}SE_{1:m}$ AND $SE^{-1}_{1:m}RB_{1:n}$

This section illustrates how frames can be used to characterize permutations performed by relatively complex networks with more than n stages. It is first shown that the balanced matrices that pass the network $RB_{1:n}SE_{1:m}$, $m \geq 0$, are identified by the frame $F^{sa}_{1:n}F^{*}_{1:m}$ (Theorem V.1), then it is shown that $RB_{1:n}SE_{1:m}$ is functionally and topologically equivalent to $SE^{-1}_{1:m}RB_{1:n}$ (Theorem V.2). Hence, any balanced matrix passing $RB_{1:n}SE_{1:m}$ also passes $SE^{-1}_{1:m}RB_{1:n}$, and vice versa. Theorem V.1 also shows how the addition of a SE stage to the right of $RB_{1:n}SE_{1:m}$ affects the type of permutations realized by the network. Theorem V.2 proves that the addition of a SE stage to the right of $RB_{1:n}SE_{1:m}$ is equivalent to the addition of an inverse SE stage to the left of $SE^{-1}_{1:m}RB_{1:n}$. All the proofs are provided in the Appendix.

V.1. Balanced Matrices and Shuffle-Exchange Networks

Linial and Tarsi [2] have shown how balanced matrices can be used to determine the number of SE stages (or the number of passes through a single SE stage) necessary to realize a given permutation. Lemma V.1 below restates their result using the notation and assumptions of this paper.

Lemma V.1. [2] Let $M_{N\times m}$ and $C_{N\times k}$ be balanced matrices such that $M_{N\times m} = [I_{N\times n}\ C_{N\times k}]$, $k \geq 1$ and $n+k=m$. The network $SE_{N\times k}$ realizes the permutation represented by $M_{(m+1-n):m}$.

To illustrate Lemma V.1, consider the identity permutation matrix $I_{8\times 3} = [i_1\ i_2\ i_3]$ and the balanced matrices $M_{8\times 4} = [I_{8\times 3}\ i_1]$, $M_{8\times 5} = [I_{8\times 3}\ i_1\ i_2]$ and $M_{8\times 6} = [I_{8\times 3}\ I_{8\times 3}]$. Because $M_{8\times 4}$, $M_{8\times 5}$ and $M_{8\times 6}$ are balanced, the permutations

represented in binary by $[i_2\ i_3\ i_1]$, $[i_3\ i_1\ i_2]$ and $[i_1\ i_2\ i_3]$ are realized by the single-stage SE, 2-stage SE and 3-stage SE with N=8 inputs/outputs, respectively.

V.2. Permutations Realized by $RB_{1:n}SE_{1:m}$

The following theorem shows how the concatenated frame $F^{sa}_{1:n}F^{*}_{1:m}$ can be used to characterize the permutations realized by $RB_{1:n}SE_{1:m}$.

Theorem V.1. A balanced matrix $D_{1:(n+m)}$, $m \geq 0$, fits the frame $F^{sa}_{1:n}F^{*}_{1:m}$ if and only if $D_{1:(n+m)}$ passes the network $RB_{1:n}SE_{1:m}$. Moreover, $RB_{1:n}SE_{1:m}$ realizes the permutation represented by $D_{(m+1):(n+m)}$.

V.3. Permutations Realized by $SE^{-1}_{1:m}RB_{1:n}$

It is shown that the network $SE^{-1}_{1:m}RB_{1:n}$ constructed by appending the network $SE^{-1}_{1:m}$ to the left of $RB_{1:n}$ is functionally and topologically equivalent to the network $RE_{1:n}SE_{1:m}$ constructed by appending $SE_{1:m}$ network to the right of $RB_{1:n}$. Also, because $RB_{1:n}$ is functionally and topologically equivalent to $BE_{1:n}$, Theorem V.2 remains valid when $RB_{1:n}$ is replaced by $BE_{1:n}$.

Theorem V.2. The network $RB_{1:n}SE_{1:m}$, $m \geq 1$, is topologically and functionally equivalent to the network $SE^{-1}_{1:m}RB_{1:n}$.

VI. NEW PROOFS FOR REARRANGEABILITY OF BENES AND THREE-STAGE CLOS NETWORKS

Rearrangeability of Benes and three-stage Clos networks is proven in [7,13] using the Slepian-Duguid theorem which applies only to symmetric networks. In this section, new simpler proofs are provided for rearrangeability of these networks using balanced matrices and the properties of graph theory. These proofs directly lead to routing algorithms [19] and provide an insight into the proofs of Section VII that identify the permutations realized by subnetworks of the Benes network. In what follows, some known results from [2] and definitions used in the proofs are presented first. Lemma VI.1 from [2] is self-explanatory.

Lemma VI.1. [2] For $n \geq 2$, let A and B be two $N\times(n-1)$ balanced matrices. Then there exists a column vector x such that both $[A\ x]$ and $[x\ B]$ are balanced matrices.

Note that, when the order of columns in a balanced matrix with at most n columns is changed, the matrix remains balanced. Therefore, the position of x in the matrices A and B in Lemma VI.1 is immaterial. Because the possible choices of vector x increase

as the number of columns of A or B is reduced, Lemma VI.1 remains valid when A and B have less than $n-1$ columns.

Some properties of balanced matrices can be captured by graphs. Therefore, some basic definitions of graph theory are given below. A graph $G=(V,E)$ consists of a set of vertices V and a set of edges E, each of which is a pair of vertices. The *union of two graphs* $G_1=(V,E_1)$ and $G_2=(V,E_2)$ is the graph $G=G_1 \cup G_2=(V,E_1 \cup E_2)$. In other words, an edge is present in $G=G_1 \cup G_2$ if and only if it is present in either G_1 or G_2. A subset M of edges in a graph G is called independent or a *matching* if no two edges of M have a vertex in common. A matching M is said to be a *perfect matching* if it covers all vertices of G. More extended discussion of these basic concepts can be found in [3,4].

Definition VI.1. (Perfect matching graph of a matrix): Let A be an $N \times k$ ($1 \le k \le n-1$, $n \ge 2$) balanced matrix. A *perfect matching graph of* A, denoted by PG_A, is a graph whose vertices are in one-to-one correspondence with the rows of A, have degree one and vertices v_i and v_j are joined by an edge only if the ith row and jth row of A are identical.

If the number of columns in a balanced matrix $A_{N \times k}$ is less than $n-1$ (i.e., if $k<n-1$), then its perfect matching graph is not unique because each distinct row in A appears 2^{n-k} times. If $k=n-1$, then PG_A is unique because each distinct row in A appears twice. As an example, consider the balanced matrix $F_{8\times 2}$ presented just after Definition II.4. Its perfect matching graph is unique and shown in Figure VI.1a.

Definition VI.2. (Labeling): 2*–labeling* or 2*–coloring* of a graph is the assignment of integers 0 and 1 to its vertices such that the labels of the vertices incident with an edge are different.

Fact VI.1. [2]. The union of two perfect matching graphs with the same set of vertices is a union of disjoint even cycles and, therefore, it can be 2-labeled.

Definition VI.3. (Perfect matching graph of a frame column): Let f_m denote a column of a frame $F_{N \times k}$. A *perfect matching graph of* f_m, denoted by PG_{f_m}, is a graph whose vertices are in one-to-one correspondence with the row labels of $F_{N \times k}$, have degree one and vertices v_i and v_j are joined by an edge only if i and j belong to the same block of f_m.

Example VI.1. One possible perfect matching graph for frame column f_2^6 in Figure III.1b is shown in Figure VI.1a. The graph in Figure VI.1b is the unique perfect

matching graph of frame column f_1^4 in Figure III.1b.

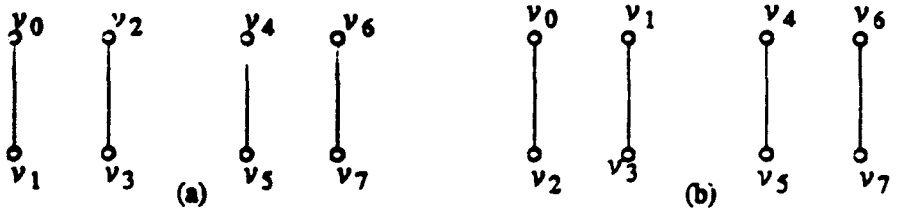


Figure VI.1. (a) The perfect matching graph of $F_{8\times2}$; it is also one possible perfect matching graph for f_2^4 shown in Figure III.1b.
(b) The unique perfect matching graph for f_1^4 shown in Figure III.1b.

From Definition VI.3 and Example VI.1, it is clear that the perfect matching graph of the frame column that consists of only the blocks of size two is unique and is also a perfect matching graph for all the other columns in the same frame.

Let the black box, called $P(N!)$ and shown in Figure VI.2, denote a rearrangeable (permutation) network on N elements, i.e., it realizes all N! distinct permutations in a single pass.

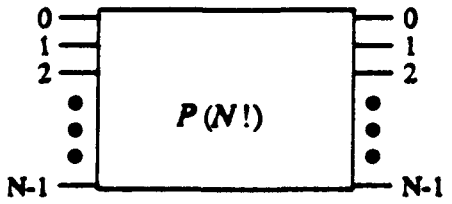


Figure VI.2. A black box $P(N!)$ which realizes all $N!$ permutations.

This black box $P(N!)$ can be expanded recursively using Algorithm CONS_BENES presented below until all of its black boxes are identical to (2×2) switching boxes (SBs), each of which can be set both straight and cross. This expansion results in the Benes network. Algorithm CONS_BENES substitutes the three-stage Clos network with R inputs/outputs, denoted by $CS_{R\times3}$ and shown in Figure VI.3, for the black box $P(R!)$.

Algorithm CONS_BENES

Input: A black box called $P(N!)$.

Output: Benes Network

Step 1. Let R be an integer variable and be initialized to N. Relabel the black box $P(N!)$ by $P(R!)$ and let BS denote a network consisting of $P(R!)$.

Step 2. Replace each and every black box called $P(R!)$ of BS by $CS_{R\times3}$ shown in Figure VI.3.

Step 3. If all the SBs of BS are (2×2), then call BS Benes network and stop; otherwise first relabel each of its non-(2×2) SBs by $P(R!)$ and halve the value of R, then go to Step 2.

Using the notions of balanced matrices and frames, it is first shown in the following theorem that $CS_{R\times3}$ is functionally equivalent to $P(R!)$. Then, it follows that the Benes network constructed by Algorithm CONS_BENES is rearrangeable because, due to the recursive structure of the algorithm, only the correctness of Step 2 needs to be proven.

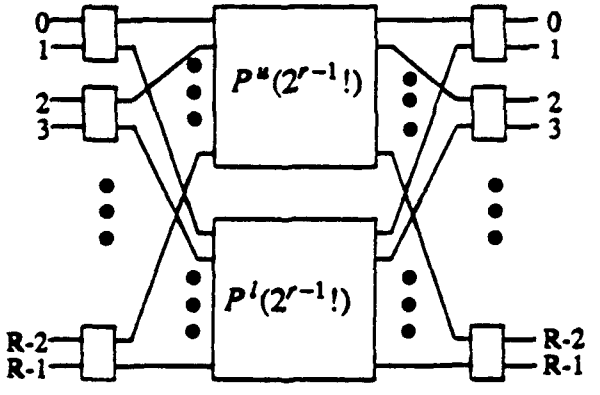


Figure VI.3. Three-stage Clos network with R inputs/outputs which is denoted by $CS_{R\times3}$, where $R=2^r$.

Theorem VI.1. Three-stage Clos network with R inputs is rearrangeable.

Proof. As it is shown in Fig. VI.4, the network $CS_{R\times3}$ is composed of three components, namely, *a*) an inverse SE stage with 2^r inputs/outputs, *b*) a pile of two permutation networks $P^u(2^{r-1}!)$ and $P^l(2^{r-1}!)$, and *c*) a SE stage with 2^r inputs/outputs. It is assumed in this proof that, unless otherwise stated, any balanced matrix has $R=2^r$ rows. Recall that $P(2^r!)$ refers to a rearrangeable network on 2^r elements. Because $P(2^r!)$ passes any balanced matrix $B_{1:r}$ corresponding to a permutation on 2^r elements, $CS_{R\times3}$ must also pass $B_{1:r}$ in order to state that $CS_{R\times3}$ is functionally equivalent

to $P(2^r!)$.

It is now shown that the inverse SE stage with 2^r inputs/outputs partitions $B_{1:r}$ into $B^u_{2^r-1\times r}$ and $B^l_{2^r-1\times r}$ such that the submatrices $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$ are balanced, where $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$ are the first $(r-1)$ columns of $B^u_{2^r-1\times r}$ and $B^l_{2^r-1\times r}$, respectively. Both $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$ pass the permutation network $P(2^{r-1}!)$ because it realizes any permutation on 2^{r-1} elements. Because the control bits of each SB must constitute the set $\{0,1\}$ to avoid conflict, any vector that fits f^a_1 can be used as the vector of control bits of the SBs of the inverse SE stage. Let the perfect matching graph of f^a_1 denote a graph with R vertices such that the vertices v_{2j} and v_{2j+1}, $0 \le j \le 2^{r-1}-1$, are connected by an edge, where v_{2j} and v_{2j+1} correspond to the $2j$th and $(2j+1)$th rows of f^a_1, respectively. Let x be a column vector obtained by a 2-labeling of the union of the perfect matching graphs of f^a_1 and $B_{1:(r-1)}$. By Fact VI.1, the matrix $[x\ B_{1:(r-1)}]$ is balanced. This implies that x "partitions" the balanced matrix $B_{1:(r-1)}$ into two balanced submatrices $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$ in such a way that row i of $B_{1:(r-1)}$ belongs to $B^u_{2^r-1\times(r-1)}$ if the ith entry of x equals zero, and belongs to $B^l_{2^r-1\times(r-1)}$ otherwise, where $0 \le i \le 2^r-1$. Without loss of generality, assume that the SBs of the inverse SE stage with 2^r inputs/outputs are labeled in ascending order starting with 0 and that the control bit for the ith input is the ith entry of x. So, when the $2j$th and $(2j+1)$th entries of x are used as control bits for the jth SB of the inverse SE stage, no conflict occurs and, hence, the matrix $B_{1:(r-1)}$ is partitioned into $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$. Because both $P^u(2^{r-1}!)$ and $P^l(2^{r-1}!)$ can pass any balanced matrix of order $2^{r-1}\times(r-1)$, the matrices $B^u_{2^r-1\times(r-1)}$ and $B^l_{2^r-1\times(r-1)}$ pass $P^u(2^{r-1}!)$ and $P^l(2^{r-1}!)$, respectively.

In order for $B_{1:r}$ to pass $CS_{R\times 3}$, $CS_{R\times 3}$ must send its ith input to the output whose value equals $B_{1:r}(i)$. So far, this proof showed that $CS_{R\times 3}$ sends its ith input with the row $B_{1:r}(i)$ to either the hth output of $P^u(2^{r-1}!)$ or the hth output of $P^l(2^{r-1}!)$, where h equals the value of $B_{1:(r-1)}(i)$. Because $B_{1:r}$ is a balanced matrix, the last entries of the routing tags of the rows that are sent to the jth outputs of $P^u(2^{r-1}!)$ and $P^l(2^{r-1}!)$ constitute the set $\{0,1\}$. Due to the fact that the third component of $CS_{R\times 3}$ is an SE stage, the rows that are sent to the jth outputs of $P^u(2^{r-1}!)$ and $P^l(2^{r-1}!)$ enter the jth SB of the SE stage. Because the connections of the SE stage implement the perfect shuffle permutation and the last entries of the routing tags of the rows entering a SB constitute the set $\{0,1\}$, no conflict occurs in the SBs. It follows that $CS_{R\times 3}$ sends its ith input to the output whose value equals $B_{1:r}(i)$. Therefore, the theorem holds. □

Corollary VI.1. The Benes network obtained by Algorithm CONS_BENES is rearrangeable.

Proof. Because Steps 1 and 3 of Algorithm CONS_BENES are relabelings and the network is constructed recursively, it suffices to show that $CS_{R\times 3}$ is functionally equivalent to $P(2^r!)$. Because this is proven in Theorem VI.1, the corollary holds. □

VII. PERMUTATIONS REALIZED BY $BS_{(n-r):(2n-1)}$

Recall that Benes network can be considered as being $BE_{N\times(n-1)}RB_{N\times n}$. Theorem IV.1 identified the permutations passing $RB_{N\times n}$ in the sense that a balanced matrix $D_{N\times n}$ passes $RB_{N\times n}$ if and only if $D_{N\times n}$ fits $F^{sa}_{N\times n}$. Likewise, the following theorem and corollary determine the set of permutations that pass the network $BS_{(n-r):(2n-1)}$ which consists of the subnetwork $BS_{(n-r):(n-1)}$ followed by $RB_{N\times n}$, where $1 \le r \le n-1$. (Recall that $IN_{x:y}$ denotes the stages x through y of an IN and that $IN_{x:y}$ refers to a nil network if $x>y$). The permutations that pass $BS_{(n-r):(2n-1)}$ are characterized by the frames defined next. This characterization illustrates how frames can be used to gain insight into why the Benes network is rearrangeable. All the proofs are provided in the Appendix. An example is presented to illustrate the results of these proofs. For $N=16$, this example clearly shows how the addition of the stage BS_{n-r-1} to the left of $BS_{(n-r):(2n-1)}$ converts the frame that corresponds to $BS_{(n-r):(2n-1)}$ into a new frame that corresponds to the resulting network.

Definition VII.1. ($F^{sa,r}_{1:k}$): The frame $F^{sa,r}_{1:k}$, $r \in \{0,1,\ldots,k-1\}$ and $k \in \{1,2,\ldots,n\}$, is a frame $<\beta,\gamma,P>$ where

$$\beta(i) = \begin{cases} r+1 & \text{if } 1 \le i \le r+1 \\ i & \text{if } r+1 < i \le k, \end{cases}$$

γ is the identity permutation on the set $\{0,1,\ldots,N-1\}$ and

$$P_i = \begin{cases} P^*_{r+1} & \text{if } 1 \le i \le r+1 \\ P^*_i & \text{if } r+1 < i \le k. \end{cases}$$

Note that $F^{sa,0}_{1:k}$ and $F^{sa,n-1}_{1:k}$ are identical to $F^{sa}_{1:k}$ and $F^{*}_{1:k}$, respectively. As examples of $F^{sa,r}_{1:k}$, the frames $F^{sa,0}_{1:4}$, $F^{sa,1}_{1:4}$, $F^{sa,2}_{1:4}$ and $F^{sa,3}_{1:4}$ for $N=16$ are illustrated in Figure VII.1.

Theorem VII.1. Consider the frame $F^{sa,r}_{N\times n}$, $0 \le r \le n-1$. Let S be a pile of 2^{n-r-1} copies of a rearrangeable network $P(2^{r+1}!)$. Let T be an IN that consists of the network S followed by $RB_{(r+2):n}$. A balanced matrix $D_{N\times n}$ fits $F^{sa,r}_{N\times n}$ if and only if $D_{N\times n}$ passes T.

Corollary VII.1. A balanced matrix $D_{N\times n}$ fits the frame $F^{sa,r}_{N\times n}$ if and only if $D_{N\times n}$ passes the network $BS_{(n-r):(2n-1)}$, where $0 \le r \le n-1$.

Example VII.1. Let $N=16$ and $n=4$. The frames $F^{sa\,0}_{16\times4}$ $F^{sa\,1}_{16\times4}$ $F^{sa\,2}_{16\times4}$ and $F^{sa\,3}_{16\times4}$ are shown in Figure VII.1. By Theorem IV.1, all balanced matrices that fit $F^{sa}_{16\times4}$ pass $RB_{:16\times4}$. If the stage BE_3 is added to the left of $RB_{1:4}$, the network $BS_{3:7}$ shown in Figure VII.2a is obtained. While $RB_{1:4}$ passes all balanced matrices that fit $F^{sa\,0}_{16\times4}$ (the same as $F^{sa}_{16\times4}$), a balanced matrix $D_{1:4}$ passes $BS_{3:7}$ if and only if $D_{1:4}$ fits $F^{sa\,1}_{16\times4}$. If the stage BE_2 is added to the left of $BS_{3:7}$, then $BS_{2:7}$ shown in Figure VII.2b is obtained. A balanced matrix $D_{1:4}$ passes $BS_{2:7}$ if and only if $D_{1:4}$ fits $F^{sa\,2}_{16\times4}$. If the stage BE_1 is added to the left of $BS_{2:7}$, then Benes network, $BS_{1:7}$, shown in Figure II.4 is obtained. It is obvious that a balanced matrix $D_{1:4}$ passes $BS_{1:7}$ if and only if $D_{1:4}$ fits $F^{sa\,3}_{16\times4}=F^{*}_{16\times4}$. Notice that, when the stage BE_j, $1\le j\le n-1$, is added to the left of $BE_{(j+1):(n-1)}RB_{1:n}$, the subnetwork $BE_{j:(n-1)}RB_{1:(n-j+1)}$ becomes a pile of 2^{j-1} copies of Benes network with 2^{n-j+1} inputs/outputs and $2n-2j+1$ stages. Because Benes network with 2^{n-j+1} inputs/outputs and $2n-2j+1$ stages is a rearrangeable network, it corresponds to the universal frame with 2^{n-j+1} rows and $n-j+1$ columns. Therefore, the first $n-j+1$ columns of $F^{sa\,n-j}_{16\times4}$ is a pile of 2^{j-1} copies of the universal frame with 2^{n-j+1} rows and $n-j+1$ columns. End of example.

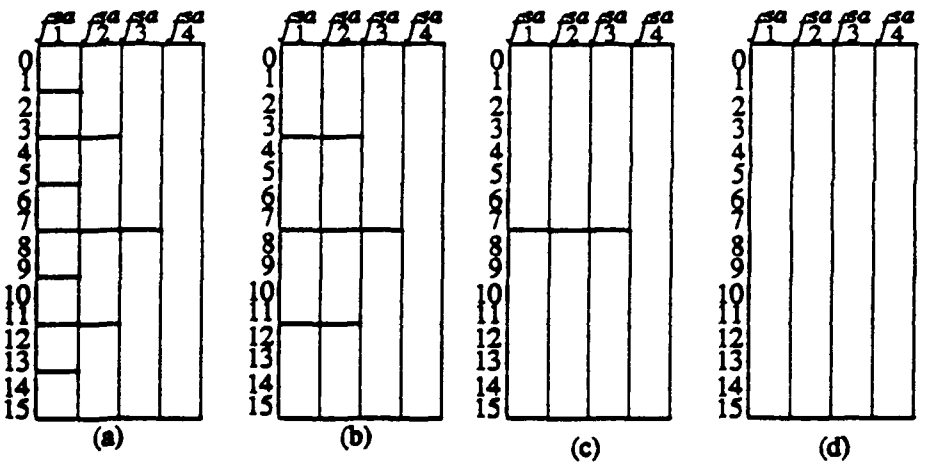
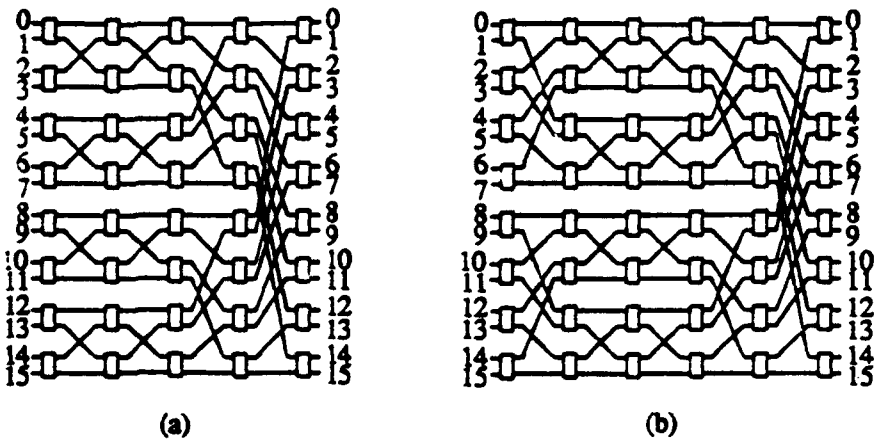


Figure VII.1. (*a*) $F^{sa\,0}_{16\times4}$, (*b*) $F^{sa\,1}_{16\times4}$, (*c*) $F^{sa\,2}_{16\times4}$ and (*d*) $F^{sa\,3}_{16\times4}$.



(a) (b)

Figure VII.2. (a) $BS_{3:7}$ (BE_3 followed by $RB_{1:4}$). (b) $BS_{2:7}$ ($BE_{2:3}$ followed by $RB_{1:4}$).

VIII. CONCLUSIONS

In this paper, a new approach has been developed to characterize permutations realized by some frequently used networks. The concept of *frame* has been introduced and different frames have been illustrated. It is simple to check whether a given permutation is realized by a given network once the corresponding frame and the output interconnection pattern are known.

The permutations of the following three classes of networks have been characterized: the class of k-stage baseline-type networks that are topologically equivalent to the k-stage baseline network, the class of those networks that are constructed by appending shuffle-exchange stages to the left or right of a baseline-type network, and the class of those networks that form a part of Benes network.

The proof that Benes network is rearrangeable was first presented in [7]. This proof is based on the Slepian-Duguid theorem which applies only to symmetric networks. In this paper, a new simple proof has been presented for rearrangeability of Benes and three-stage Clos networks using the notion of balanced matrices and graph theory. The technique used in this proof can also be applied to nonsymmetric networks.

In practice, the results presented in this paper can be used to design networks that realize classes of permutations that fit the same frame. In addition, engineers and/or compilers may use frames to test if the corresponding networks realize a given permutation. Debuggers and programming environment can also use frames to detect when and why a permutation cannot be realized by the network. The definitions, theorems and lemmata that are presented in this paper to characterize the permutations realized in the aforementioned networks can also be used to address the issues of routing and counting permutations. But, to limit the size of this paper, these issues are addressed in

[19,28].

It is clear that frames, as defined in this paper, cannot characterize the permutations of every network. Conceivably, extensions of the definitions may be possible to characterize a larger class of networks. In particular, the concepts should be extensible to networks not considered in this paper including those constructed with $(k\times k)$ switches for $k>2$. Future research will address these issues.

IX. APPENDIX

Proof Theorem IV.1: $(\rightarrow)$ It is shown that if $D_{1:k}$ fits $F^{sa}_{1:k}$ then $D_{1:k}$ passes $RB_{1:k}$. Proof is by induction on k. Also, it is proven that $RB_{1:k}$ sends its ith input to its jth output, where j is equal to the sum of $\left\lfloor i/2^k \right\rfloor \times 2^k$ and the value of $D_{1:k}(i)$.

Basis Step: Let $k=1$. Label the SBs of RB_1 in ascending order starting with 0. (Recall that RB_1 refers to the first stage of a reverse baseline network with N inputs/outputs). By definition, f^{sa}_1 contains 2^{n-1} blocks of size 2 each. The fact that $D_{1:k}$ fits $F^{sa}_{1:k}$ implies that d_1 fits f^{sa}_1. Therefore, the $2r$th and $(2r+1)$th entries of d_1 constitute the set $\{0,1\}$, where $0 \le r \le 2^{n-1}-1$. Hence, when the $2r$th and $(2r+1)$th entries of d_1 are used as the control bits to set the rth SB of RB_1, no conflict occurs and RB_1 sends its ith input to its jth output, where j is equal to the sum of $\left\lfloor i/2 \right\rfloor \times 2$ and the value of the ith entry of d_1, where $0 \le i \le N-1$. (Recall that, if the control bit of the routing tag of an input equals zero, then the input is sent to the upper output of the SB that it enters; otherwise it is sent to the the lower output of the SB).

Induction Step: Assume that, for $2 \le k \le n$, if $D_{1:(k-1)}$ fits $F^{sa}_{1:(k-1)}$, then $D_{1:(k-1)}$ passes $RB_{1:(k-1)}$ and $RB_{1:(k-1)}$ sends its ith input to its jth output, where j is equal to the sum of $\left\lfloor i/2^{k-1} \right\rfloor \times 2^{k-1}$ and the value of $D_{1:(k-1)}(i)$. Now, show that, if $D_{1:k}$ fits $F^{sa}_{1:k}$, then $D_{1:k}$ passes $RB_{1:k}$ and $RB_{1:k}$ sends its ith input to its jth output, where j is equal to the sum of $\left\lfloor i/2^k \right\rfloor \times 2^k$ and the value of $D_{1:k}(i)$.

The frame $F^{sa}_{1:m}$, $(m=k-1,k)$, can be considered as being composed of 2^{n-m} copies of $F^{sa}_{2^m\times m}$ in parallel if the row labels of the αth, $0 \le \alpha \le 2^{n-m}-1$, $F^{sa}_{2^m\times m}$ consists of the numbers $(\alpha\times 2^m)$ to $[(\alpha+1)\times 2^m-1]$ inclusive. Let $F^{\alpha}_{2^m\times m}$ denote the αth $F^{sa}_{2^m\times m}$. $RB_{1:m}$ can also be considered as being the pile of 2^{n-m} distinct $RB_{2^m\times m}$s. Label these $RB_{2^m\times m}$s in ascending order starting with 0 at the top and denote the αth one by $RB^{\alpha}_{2^m\times m}$. By hypothesis, $D_{1:m}$ fits $F^{sa}_{1:m}$. Let $D^{\alpha}_{2^m\times m}$ denote the submatrix of $D_{1:m}$ that

fits $F^{\alpha}_{2^{n-k}\times n}$. Thus, the induction hypothesis also implies that $D^{\alpha}_{2^{k-1}\times k-1}$ (which fits $F^{\alpha}_{2^{k-1}\times k-1}$) passes $RB^{\alpha}_{2^{k-1}\times k-1}$ and that $RB^{\alpha}_{2^{k-1}\times k-1}$ sends its pth input to the output whose value is equal to the value of $D^{\alpha}_{2^{k-1}\times k-1}(p)$, where $0 \le p \le 2^{k-1}-1$.

Let $F^{\alpha_1}_{2^{k-1}\times k-1}$ and $F^{\alpha_2}_{2^{k-1}\times k-1}$ denote the $2l$th and $(2l+1)$th $F^{\alpha}_{2^{k-1}\times k-1}$s, respectively, where $0 \le l \le 2^{n-k}-1$. Similarly, let $D^{\alpha_1}_{2^{k-1}\times k-1}$ and $D^{\alpha_2}_{2^{k-1}\times k-1}$ denote the $2l$th and $(2l+1)$th $D^{\alpha}_{2^{k-1}\times k-1}$s. Likewise, assume that $RB^{\alpha_1}_{2^{k-1}\times k-1}$ and $RB^{\alpha_2}_{2^{k-1}\times k-1}$ denote the $2l$th and $(2l+1)$th $RB^{\alpha}_{2^{k-1}\times k-1}$s.

Because $D^{\alpha}_{2^{k-1}\times(k-1)}$ is a balanced matrix of order $2^{k-1}\times(k-1)$, it has 2^{k-1} distinct rows. Therefore, the matrix

$$H = \begin{bmatrix} D^{\alpha_1}_{2^{k-1}\times(k-1)} \\ D^{\alpha_2}_{2^{k-1}\times(k-1)} \end{bmatrix}$$

contains 2^{k-1} distinct rows, each being repeated twice. Assume that the rows of H are partitioned into 2^{k-1} classes, each of which contains 2 identical rows, that is, each class contains the two copies of a distinct row of H. After adding a column permutation of length 2^k to the right of H, call the resultant matrix $D^{\beta}_{2^{k-1}\times(k)}$. This implies that the number of the entries of the rows of a class is incremented by 1. In order for $D^{\beta}_{2^{k-1}\times(k)}$ to fit $F^{\alpha}_{2^{k-1}\times k}$ the k*th* entries of the rows of each class of H must constitute the set $\{0,1\}$, which is true because $D_{1:k}$ fits $F^{sa}_{1:k}$ by the induction hypothesis.

By definition, the kth stage of reverse baseline, RB_k, consists of a pile of 2^{n-k} copies of the SE stage with 2^k inputs/outputs. Assume that the network consisting of the pile of two networks $RB^{\alpha_1}_{2^{k-1}\times(k-1)}$ and $RB^{\alpha_2}_{2^{k-1}\times(k-1)}$ followed by the SE stage with 2^k inputs/outputs is called $RB^{\beta}_{2^{k-1}\times k}$. Because $RB^{\alpha}_{2^{k-1}\times k-1}$ sends its pth input to the output whose value is equal to the value of $D^{\alpha}_{2^{k-1}\times k-1}(p)$, the first $(k-1)$ entries of the row that is sent to the pth output of the network $RB^{\alpha_1}_{2^{k-1}\times(k-1)}$ is the same as the first $(k-1)$ entries of the row that is sent to the pth output of the network $RB^{\alpha_2}_{2^{k-1}\times(k-1)}$. The kth entries of those two rows sent to the pth outputs of $RB^{\alpha_1}_{2^{k-1}\times(k-1)}$ and $RB^{\alpha_2}_{2^{k-1}\times(k-1)}$ constitute the set $\{0,1\}$ because $D^{\beta}_{2^{k-1}\times k}$ fits the frame $F^{\alpha}_{2^{k-1}\times k}$ by induction hypothesis. Because the rows that are sent to the pth outputs of $RB^{\alpha_1}_{2^{k-1}\times(k-1)}$ and $RB^{\alpha_2}_{2^{k-1}\times(k-1)}$ enter the pth SB of the SE stage following these networks such that the kth entries of these rows are the control bits for the SB, no conflict occurs in the pth SB. This amounts to stating that $RB^{\beta}_{2^{k-1}\times k}$ sends its hth input to the output whose value is equal to the value of $D^{\beta}_{2^{k-1}\times k}(h)$, where $0 \le h \le 2^k-1$. Therefore, the balanced matrix $D_{1:k}$ passes $RB_{1:k}$ and $RB_{1:k}$ sends its ith input to its jth output, where $0 \le i \le N-1$ and j is equal to the sum of $\left\lfloor i/2^k \right\rfloor \times 2^k$ and the value of $D_{1:k}(i)$.

($\leftarrow$) It is shown that, if $D_{1:k}$ passes $RB_{1:k}$, then $D_{1:k}$ fits $F^{sa}_{1:k}$. Proof is by induction on k.

Basis Step: Let k=1. The fact that d_1 passes RB_1 implies that no conflict occurs in the SBs of RB_1 when the ith entry of d_1 is used as the control bit for the ith input of RB_1 in setting its rth SB. Because the control bits of the rth SB of RB_1 constitute the set {0,1} and fit the rth block of f_1^a, d_1 fits f_1^a.

Induction Step: Assume that the theorem holds for $k-1$. It is shown that it also holds for k, where $2 \le k \le n$.

By induction hypothesis, if $D^{\alpha}_{2^{n-1}\times k-1}$ passes $RB^{\alpha}_{2^{n-1}\times k-1}$ fits $F^{\alpha}_{2^{n-1}\times k-1}$. Notice that the last stage of $RB^{\beta}_{2^n\times k}$ is the SE stage with 2^k inputs/outputs. Recall that the network consisting of the pile of two networks $RB^{\alpha_1}_{2^{n-1}\times(k-1)}$ and $RB^{\alpha_2}_{2^{n-1}\times(k-1)}$ followed by the SE stage with 2^k inputs/outputs is called $RB^{\beta}_{2^n\times k}$. As it is also explained above, the rows that are sent to the pth outputs of $RB^{\alpha_1}_{2^{n-1}\times(k-1)}$ and $RB^{\alpha_2}_{2^{n-1}\times(k-1)}$ enter the pth SB of the SE stage that follows these networks. If $D^{\beta}_{2^n\times k}$ passes $RB^{\beta}_{2^n\times k}$, then the kth entries of the rows of a class of H must constitute the set {0,1} to avoid having a conflict in the pth SB. Therefore, $D^{\beta}_{2^n\times k}$ fits $F^{\beta}_{2^n\times k}$. It follows that $D_{1:k}$ fits $F^{sa}_{1:k}$. □

Proof of Corollary IV.1: (→) Let Φ be topologically equivalent to $RB_{1:k}$. When interconnection networks are modeled by directed graphs in which vertices represent the switches and edges the links, two networks are said to be topologically equivalent if the graphs representing them are isomorphic. Two graphs G and H are said to be isomorphic if there exist bijections from the vertices and edges of G to the vertices and edges of H, respectively such that the relationship of adjacency is preserved. So, if two networks are topologically equivalent to each other, one of them can be made identical to the other network by relabeling the inputs and/or outputs. This implies that Φ can be made identical to $RB_{1:k}$ by relabeling the inputs and/or outputs of Φ, and vice versa. Because (1) $F^{sa}_{1:k}$ corresponds to $RB_{1:k}$ such that there exists a one-to-one correspondence between the row labels of $F^{sa}_{1:k}$ and $RB_{1:k}$ (Theorem IV.1), (2) the only difference between $F^{sa}_{1:k}$ and an a-type frame $F^{a}_{1:k}$ is the order of their row labels, and (3) Φ is topologically equivalent to $RB_{1:k}$, there exists an a-type frame $F^{a}_{1:k}$ corresponding to Φ such that no conflict occurs in the switches of Φ when the contents of the ith row, $0 \le i \le N-1$, of $F^{a}_{1:k}$ are used as the routing tag for the ith input of Φ.

(←) Let γ denote the vector of input labels of Φ such that the ith entry of γ equals the ith input label of Φ. Let $F^{a}_{1:k}(\gamma)$ denote the frame corresponding to Φ such that the ith entry of γ equals the ith row label of the frame. By definition of "correspondence" (Definition III.8), no conflict occurs in the switches of Φ when the contents of the ith row of $F^{a}_{1:k}(\gamma)$ are used as the routing tag for the ith input of Φ. Note that there exists a one-to-one correspondence between the input labels of Φ and the row labels of $F^{a}_{1:k}(\gamma)$. Therefore, when both the ith row label of $F^{a}_{1:k}(\gamma)$ and the ith input label of Φ are replaced by the integer i, the resulting frame $F^{sa}_{1:k}$ and network still remain correspondent to each other. By Theorem IV.1, $F^{sa}_{1:k}$ corresponds to $RB_{1:k}$. It follows

that Φ can be converted to $RB_{1:k}$ by relabeling the input and/or output labels of Φ. Thus, Φ is topologically equivalent to $RB_{1:k}$. □

Definition IX.1. (forward-routing, reverse-routing): Given an $IN_{N\times k}$ and a setting of its SBs that realizes $h : i \rightarrow h(i)$, *forward-routing* of a matrix A means that $A(i)$ is sent from input i to output $h(i)$, where $0 \le i \le N-1$. Likewise, *reverse-routing* of A means that $A(i)$ is sent from the output i to the input $h^{-1}(i)$. The matrix $A^F = A(h^{-1}(i))$, $i = 0, 1, \ldots, N-1$, is obtained by forward-routing of A. Similarly, the matrix $A^R = A(h(i))$, $i = 0, 1, \ldots, N-1$, is obtained by reverse-routing of A.

Proof of Corollary IV.3: Because the network Π is a k-stage baseline-type network, it is topologically equivalent to $RB_{1:k}$. This implies that $RB_{1:k}$ can be made identical to Π by relabeling its inputs and/or outputs. Because relabeling the inputs (respectively, outputs) of $RB_{1:k}$ is equivalent to adding an interconnection pattern to the left (respectively, right) of $RB_{1:k}$, there exist two interconnection patterns IP_{in} and IP_{out} such that Π is topologically and functionally equivalent to $IP_{in}RB_{1:k}IP_{out}$.

($\rightarrow$) Assume that $D_{1:k}$ fits $F^a_{1:k}(\alpha_{in}^{-1})$. It is shown that the network Π realizes the permutation $\alpha_{in}.\mu.\alpha_{out}$.

Adding the interconnection pattern IP_{in} to the left of $RB_{1:k}$ is equivalent to relabeling the ith input of $RB_{1:k}$ by $\alpha_{in}^{-1}(i)$. Because the only difference between two a-type frames with k columns is the order of their row labels and IP_{out} is just an interconnection pattern, it follows from Theorem IV.1 that $D_{1:k}$ passes Π. By Definition IX.1, when $D_{1:k}$ is forward-routed through the interconnection pattern IP_{in}, $D_{1:k}$ is mapped to $D^*_{1:k} = D_{1:k}(\alpha_{in}^{-1}(i))$, $i = 0, 1, \ldots, N-1$. By Theorem IV.3, the subnetwork $RB_{1:k}$ of Π realizes the permutation μ represented by $[I_{1:n-k}\ D^*_{1:k}]$. Therefore, the network Π realizes the permutation $\alpha_{in}.\mu.\alpha_{out}$.

($\leftarrow$) Assume that the network Π realizes the permutation $\alpha_{in}.\mu.\alpha_{out}$. It is shown that $D_{1:k}$ fits $F^a_{1:k}(\alpha_{in}^{-1})$.

The fact that Π realizes the permutation $\alpha_{in}.\mu.\alpha_{out}$ implies that the permutation μ is realized by $RB_{1:k}$ of Π. Because μ is the permutation represented by the balanced matrix $[I_{1:n-k}\ D^*_{1:k}]$ such that $D^*_{1:k}(i) = D_{1:k}(\alpha_{in}^{-1}(i))$, it follows from Theorem IV.3 that $D^*_{1:k}$ passes $RB_{1:k}$. By Definition IX.1, when $D_{1:k}$ is reverse-routed through the interconnection pattern IP_{in}, $D^*_{1:k}$ is mapped to $D_{1:k}$. Thus, $D_{1:k}$ passes $IP_{in}RB_{1:k}$. Note that the network $IP_{in}RB_{1:k}$ is identical to the network obtained by relabeling the ith input of $RB_{1:k}$ by $\alpha_{in}^{-1}(i)$. In addition, because $F^a_{1:k}(\alpha_{in}^{-1})$ is the same as $F^{sa}_{1:k}$ except that the ith row label of $F^a_{1:k}(\alpha_{in}^{-1})$ equals $\alpha_{in}^{-1}(i)$ instead of i, $D_{1:k}$ fits $F^a_{1:k}(\alpha_{in}^{-1})$. □

Proof of Theorem V.1: ($\rightarrow$) It is shown that if $D_{1:(n+m)}$ fits $F^{sa}_{1:n}F^{*}_{1:m}$, then $D_{1:(n+m)}$ passes $RB_{1:n}SE_{1:m}$ and the permutation represented in binary by $D_{(m+1):(n+m)}$ is realized by $RB_{1:n}SE_{1:m}$.

Recall that by definition, $RB_{1:n}SE_{1:m}$ consists of $RB_{1:n}$ followed by $SE_{1:m}$. Because, by hypothesis, $D_{1:(n+m)}$ fits $F^{sa}_{1:n}F^{*}_{1:m}$, $D_{1:n}$ fits $F^{sa}_{1:n}$. $RB_{1:n}$ maps the matrix $D_{1:(n+m)}$ into the matrix denoted by $D^{*}_{1:(n+m)}$ when $D_{1:(n+m)}(i)$, $0 \le i \le N-1$, is used as the routing tag for the ith input of $RB_{1:n}$. Theorem IV.1 has shown that any balanced matrix $D_{1:n}$ fitting the frame $F^{sa}_{1:n}$ passes the network $RB_{1:n}$. So, when $D_{1:n}(i)$ is used as the routing tag for the ith input of $RB_{1:n}$, $RB_{1:n}$ sends its ith input to the output whose value equals $D_{1:n}(i)$. So, $RB_{1:n}$ maps any $D_{1:n}$ fitting the frame $F^{sa}_{1:n}$ to $I_{1:n}$. This implies that, when $D_{1:(n+m)}(i)$ is used as the routing tag for the ith input of $RB_{1:n}SE_{1:m}$, the submatrix $D^{*}_{1:n}$ of $D^{*}_{1:(n+m)}$ is the same as the identity permutation matrix $I_{1:n}$. Therefore, $D^{*}_{1:(n+m)}$ is equal to the balanced matrix $[I_{1:n}\ D^{*}_{(n+1):(n+m)}]$. By Lemma V.1, $SE_{1:m}$ realizes the permutation represented by $D^{*}_{(m+1):(n+m)}$ and no conflict occurs in the SBs of $SE_{1:m}$ when $D^{*}_{(n+1):(n+m)}(i)$ is used as the routing tag for the ith input of $SE_{1:m}$. Therefore, $D_{1:(n+m)}$ passes $RB_{1:n}SE_{1:m}$. Now, it remains to show that $RB_{1:n}SE_{1:m}$ realizes the permutation represented by $D_{(n+1):(n+m)}$.

Let the entries of $D_{1:(n+m)}(i)$ be denoted in binary by $(x^i_1 x^i_2 \cdots x^i_n \cdots x^i_{n+m})$. The fact that $D^{*}_{1:n}$ of $D^{*}_{1:(n+m)}$ is identical to $I_{1:n}$ implies that $RB_{1:n}$ of $RB_{1:n}SE_{1:m}$ sends the routing tag $D_{1:(n+m)}(i)$ to the output of $RB_{1:n}$ whose value equals the value of $(x^i_1 x^i_2 \cdots x^i_n)$. Because the jth output of $RB_{1:n}$ is the same as the jth input of $SE_{1:m}$ when $RB_{1:n}SE_{1:m}$ is considered, $D_{1:(n+m)}(i)$ is sent to the jth input of $SE_{1:m}$ by $RB_{1:n}$, where j equals $(x^i_1 x^i_2 \cdots x^i_n)$. Hence, the bit x^i_{n+p}, $1 \le p \le m$, of $(x^i_1 x^i_2 \cdots x^i_n \cdots x^i_{n+m})$ is used as the control bit to set a SB at the pth stage of $SE_{1:m}$, where $(x^i_1 x^i_2 \cdots x^i_n)$ and $(x^i_{m+1} x^i_{m+2} \cdots x^i_{n+m})$ are the addresses of the input and the destination, respectively. Due to the fact that $D_{1:(n+m)}$ passes $RB_{1:n}SE_{1:m}$ and a SE stage performs the shuffle operation followed by the exchange operation, $RB_{1:n}SE_{1:p}$ sends $D_{1:(n+m)}(i)$ to the output of $RB_{1:n}SE_{1:p}$ whose value equals $(x^i_{p+1} x^i_{p+2} \cdots x^i_{n+p})$. Therefore, the permutation represented by $D_{(m+1):(n+m)}$ is implemented by $RB_{1:n}SE_{1:m}$.

($\leftarrow$) It is shown that, if $D_{1:(n+m)}$ passes $RB_{1:n}SE_{1:m}$, then $D_{1:(n+m)}$ fits $F^{sa}_{1:n}F^{*}_{1:m}$ and $RB_{1:n}SE_{1:m}$ realizes the permutation represented by $D_{(m+1):(n+m)}$.

Because, by hypothesis, $D_{1:(n+m)}$ passes $RB_{1:n}SE_{1:m}$, the submatrix $D_{1:n}$ of $D_{1:(n+m)}$ passes $RB_{1:n}$. So, by Theorem IV.1, the submatrix $D_{1:n}$ fits $F^{sa}_{1:n}$. By definition, any column of the universal frame $F^{*}_{1:m}$ is a single block of size N. Therefore, any balanced matrix of order $(N \times m)$ fits $F^{*}_{1:m}$. It follows that $D_{(n+1):(n+m)}$ fits $F^{*}_{1:m}$. Hence, $D_{1:(n+m)}$ fits $F^{sa}_{1:n}F^{*}_{1:m}$.

The first part ($\rightarrow$) of the proof has shown that the permutation represented by $D_{(m+1):(n+m)}$ is implemented by $RB_{1:n}SE_{1:m}$ if $D_{1:(n+m)}$ fits $F^{sa}_{1:n}F^{*}_{1:m}$. Because it is

shown above that $D_{1:(n+m)}$ fits $F^{ss}_{1:n}F^{s}_{1:m}$, $RB_{1:n}SE_{1:m}$ realizes the permutation corresponding to $D_{(n+1):(n+m)}$. □

Proof of Theorem V.2: Proof is by induction on m.

Basis Step: Let $m=1$. In this step it is proven that $RB_{1:n}SE_1$ is functionally and topologically equivalent to $SE_1^{-1}RB_{1:n}$. Recall that $RB_{1:n}$ is functionally and topologically equivalent to $BE_{1:n}$. Therefore, $RB_{1:n}SE_1$ is functionally and topologically equivalent to $BE_{1:n}SE_1$. $BE_{2:n}$ consists of 2 copies of $BE_{2^{n-1}\times(n-1)}$ in parallel, while $RB_{1:(n-1)}$ consists of 2 copies of $RB_{2^{n-1}\times(n-1)}$ in parallel. Because $BE_{2^{n-1}\times(n-1)}$ is functionally and topologically equivalent to $RB_{2^{n-1}\times(n-1)}$, $BE_{2:n}$ is functionally and topologically equivalent to $RB_{1:(n-1)}$. Therefore, $BE_{1:n}SE_1$ is functionally and topologically equivalent to $BE_1RB_{1:(n-1)}SE_1$. Because the last stage of $RB_{1:n}$ is identical to the SE stage, $RB_{1:(n-1)}SE_1$ is identical to $RB_{1:n}$. Therefore, $BE_1RB_{1:(n-1)}SE_1$ is functionally and topologically equivalent to $BE_1RB_{1:n}$. Due to the fact that BE_1 is identical to the inverse SE stage, $BE_1RB_{1:n}$ is functionally and topologically equivalent to $SE_1^{-1}RB_{1:n}$. It follows that $RB_{1:n}SE_1$ is functionally and topologically equivalent to $SE_1^{-1}RB_{1:n}$.

Induction Step: Assume that, for $m \geq 2$, the theorem holds for $m-1$, and show that it also holds for m.

Because $RB_{1:n}$ is functionally and topologically equivalent to $BE_{1:n}$, $RB_{1:n}SE_{1:m}$ is functionally and topologically equivalent to $BE_{1:n}SE_{1:m}$. As it is explained in the Basis Step above, $BE_{2:n}$ is functionally and topologically equivalent to $RB_{1:(n-1)}$. Therefore, $RB_{1:n}SE_{1:m}$ is functionally and topologically equivalent to $BE_1RB_{1:(n-1)}SE_{1:m}$. Because the last stage of $RB_{1:n}$ is identical to the SE stage, $BE_1RB_{1:(n-1)}SE_{1:m}$ is identical to $BE_1RB_{1:n}SE_{1:(m-1)}$. By the induction hypothesis, $RB_{1:n}SE_{1:(m-1)}$ is functionally and topologically equivalent to $SE^{-1}_{1:(m-1)}RB_{1:n}$. So, $BE_1RB_{1:n}SE_{1:(m-1)}$ is functionally and topologically equivalent to $BE_1SE^{-1}_{1:(m-1)}RB_{1:n}$. Because BE_1 is identical to the inverse SE stage, $BE_1SE^{-1}_{1:(m-1)}RB_{1:n}$ is functionally and topologically equivalent to $SE^{-1}_{1:m}RB_{1:n}$. Thus, the theorem holds. □

Proof of Theorem VII.1: *Case 1:* Let $r=n-1$. When $r=n-1$, T consists of only a rearrangeable network $P(2^n!)$ and $F^{ss:r}_{N\times n}$ is identical to the universal frame $F^{s}_{N\times n}$. By definition, any balanced matrix of order $N\times n$ fits $F^{s}_{N\times n}$ and $P(2^n!)$ passes any balanced matrix of order $N\times n$. Therefore, a $D_{N\times n}$ fits $F^{ss:r}_{N\times n}$ if and only if $D_{N\times n}$ passes T.

Case 2: Let $r=0$. When $r=0$, $F^{ss:r}_{N\times n}$ and T are identical to $F^{ss}_{N\times n}$ and $RB_{N\times n}$, respectively. Because Theorem IV.1 shows that a $D_{N\times n}$ fits $F^{ss}_{N\times n}$ if and only if $D_{N\times n}$ passes $RB_{N\times n}$, Theorem VII.1 holds for this case.

Case 3: Let $1 \le r \le n-2$. Assume that $D_{N\times n}(i)$, $0 \le i \le N-1$, is used as the routing tag for the ith input of T.

($\rightarrow$) It is shown that, if $D_{N\times n}$ fits $F^{sa,r}_{N\times n}$, then $D_{N\times n}$ passes T.

In what follows, it is first shown that the submatrix $D_{1:(r+1)}$ of a $D_{N\times n}$ passes S. By the definition of rearrangeability, any of the 2^{n-r-1} rearrangeable networks $P(2^{r+1}!)$ of S can pass any balanced matrix of order $2^{r+1}\times(r+1)$. Label these rearrangeable networks in ascending order starting with 0. Let $P^{\alpha}(2^{r+1}!)$ denote the αth rearrangeable network $P(2^{r+1}!)$ of S, where $0 \le \alpha \le 2^{n-r-1}-1$.

Consider the universal frame $F^{*}_{2^{r+1}\times(r+1)}$. Any column of $F^{*}_{2^{r+1}\times(r+1)}$ is just a single block of length 2^{r+1}. Because a column of $F^{*}_{2^{r+1}\times(r+1)}$ requires a column vector of length 2^{r+1} to have only 2^r zeros and 2^r ones, any column of a balanced matrix of order $2^{r+1}\times(r+1)$ fits it. It follows that any balanced matrix of order $2^{r+1}\times(r+1)$ fits $F^{*}_{2^{r+1}\times(r+1)}$. Therefore, $P^{\alpha}(2^{r+1}!)$ corresponds to the universal frame $F^{*}_{2^{r+1}\times(r+1)}$. The subframe $F^{sa,r}_{1:(r+1)}$ can be considered as being a pile of 2^{n-r-1} $F^{*}_{2^{r+1}\times(r+1)}$s. Label these universal frames in ascending order starting with 0.

Partition the balanced submatrix $D_{1:(r+1)}$ of $D_{N\times n}$ into 2^{n-r-1} balanced submatrices of order $2^{r+1}\times(r+1)$ such that the set of the row indices of the αth submatrix consists of the numbers $(\alpha\times 2^{r+1})$ to $[(\alpha+1)\times 2^{r+1}-1]$ inclusive. Label these submatrices of order $2^{r+1}\times(r+1)$ in ascending order starting with 0. Denote the αth submatrix of $D_{1:(r+1)}$ by $D^{\alpha}_{1:(r+1)}$.

By hypothesis, $D_{N\times n}$ fits $F^{sa,r}_{1:n}$. This implies that $D_{1:(r+1)}$ fits $F^{sa,r}_{1:(r+1)}$. Therefore, $D^{\alpha}_{1:(r+1)}$ fits the αth $F^{*}_{2^{r+1}\times(r+1)}$. Because $P^{\alpha}(2^{r+1}!)$ is a rearrangeable network, it passes $D^{\alpha}_{1:(r+1)}$, that is, $P^{\alpha}(2^{r+1}!)$ sends its kth input to the output whose value equals $D^{\alpha}_{1:(r+1)}(k)$ where $0 \le k \le 2^{r+1}-1$. This implies that the network S sends its ith input to its jth output, where j equals the sum of $\left\lfloor i/2^{r+1} \right\rfloor \times 2^{r+1}$ and the value of the leftmost $(r+1)$ bits of the $D_{1:n}(i)$. Hence, $D_{1:(r+1)}$ passes S.

Theorem IV.1 shows that a balanced matrix $C_{1:n}$ that fits $F^{sa}_{1:n}$ passes $RB_{1:n}$. Theorem IV.1 also shows that $RB_{1:(r+1)}$ sends its ith input to its hth output, where h is equal to the sum of $\left\lfloor i/2^{r+1} \right\rfloor \times 2^{r+1}$ and the value of $C_{1:(r+1)}(i)$. Thus, the networks $RB_{1:(r+1)}$ and S send their ith inputs to their jth outputs, where j equals the sum of $\left\lfloor i/2^{r+1} \right\rfloor \times 2^{r+1}$ and the value of the ith row of the matrix passing the corresponding network that is either $RB_{1:(r+1)}$ or S. By definition, $F^{sa,r}_{(r+1):n}$ is the same as $F^{sa}_{(r+1):n}$. This implies that $F^{sa,r}_{(r+2):n}$ is also the same as $F^{sa}_{(r+2):n}$. It follows from this paragraph that the argument given in the ($\rightarrow$) part of the proof of Theorem IV.1 applies to $RB_{(r+2):n}$ of T and $F^{sa,r}_{(r+2):n}$. (If in Theorem IV.1 $RB_{1:(r+1)}$ and $F^{sa}_{1:(r+1)}$ are replaced by

S and $F^{sa,r}_{1:(r+1)}$, respectively, Theorem IV.1 becomes identical to Theorem VII.1). Therefore, $D_{N\times n}$ passes T.

($\leftarrow$) It is shown that, if $D_{N\times n}$ passes T, then $D_{N\times n}$ fits $F^{sa,r}_{N\times n}$.

First, consider the submatrix $D_{1:(r+1)}$ of $D_{1:n}$. By hypothesis, $D_{N\times n}$ passes T. This implies that $D_{1:(r+1)}$ passes S because S consists of 2^{n-r-1} copies of a rearrangeable network $P(2^{r+1}!)$ in parallel and $D^{\alpha}_{1:(r+1)}$ passes $P^{\alpha}(2^{r+1}!)$. Because any balanced matrix of order $2^{r+1}\times(r+1)$ fits a universal frame $F^{*}_{2^{r+1}\times(r+1)}$, $D^{\alpha}_{1:(r+1)}$ also fits $F^{*}_{2^{r+1}\times(r+1)}$. Recall that $F^{sa,r}_{1:(r+1)}$ can be considered as a pile of 2^{n-r-1} copies of $F^{*}_{2^{r+1}\times(r+1)}$. Therefore, $D_{1:(r+1)}$ fits $F^{sa,r}_{1:(r+1)}$.

Now, it is shown by induction on β, $1\le\beta\le n-r-1$, that $D_{1:(r+1+\beta)}$ fits $F^{sa,r}_{1:(r+1+\beta)}$, assuming that $D_{N\times n}$ passes T. (The proof presented below is analogous to part ($\leftarrow$) of the proof of Theorem IV.1).

Basis step: Let β=1. For $0\le l\le 2^{n-r-2}-1$, let $D^{\alpha_1}_{1:(r+1)}(k)$ and $D^{\alpha_2}_{1:(r+1)}(k)$ denote the $2l$th and $(2l+1)$th $D^{\alpha}_{1:(r+1)}$s, respectively. Similarly, let $P^{\alpha_1}(2^{r+1}!)$ and $P^{\alpha_2}(2^{r+1}!)$ denote the $2l$th and $(2l+1)$th rearrangeable networks of S, respectively. Because the stage $RB_{(r+2)}$ consists of a pile of 2^{n-r-2} copies of the SE stage with 2^{r+2} inputs/outputs, the subnetwork that consists of the pile of $P^{\alpha_1}(2^{r+1}!)$ and $P^{\alpha_2}(2^{r+1}!)$ is followed by the SE stage with 2^{r+2} inputs/outputs. Because $D^{\alpha}_{1:(r+1)}$ passes $P^{\alpha}(2^{r+1}!)$, $P^{\alpha}(2^{r+1}!)$ sends its kth input to its mth output, where m equals the contents of $D^{\alpha}_{1:(r+1)}(k)$. Hence, the rows that are sent to the kth outputs of $P^{\alpha_1}(2^{r+1}!)$ and $P^{\alpha_2}(2^{r+1}!)$ enter the kth SB of the succeeding SE stage with 2^{r+2} inputs/outputs. By hypothesis, $D_{N\times n}$ passes T. This implies that $D_{1:(r+2)}$ passes the network consisting of S followed by the stage RB_{r+2} without having any conflict in the SBs. Therefore, the $(r+2)$th entries of the rows that are sent to the kth outputs of $P^{\alpha_1}(2^{r+1}!)$ and $P^{\alpha_2}(2^{r+1}!)$ constitute the set $\{0,1\}$. Notice that these rows have the same first $k-1$ entries. Therefore, the $(r+2)$th entries of any two identical rows of the submatrix

$$\begin{bmatrix} D^{\alpha_1}_{1:(r+1)} \\ D^{\alpha_2}_{1:(r+1)} \end{bmatrix}$$

constitute the set $\{0,1\}$. Therefore, by definition of fit, $D_{1:(r+2)}$ fits $F^{sa,r}_{1:(r+2)}$.

Induction step: Assume that, for $2\le\beta\le n-r-1$, $D_{1:(r+\beta)}$ fits $F^{sa,r}_{1:(r+\beta)}$. Then, show that $D_{1:(r+1+\beta)}$ also fits $F^{sa,r}_{1:(r+1+\beta)}$.

Let $2\le\beta\le n-r-1$. By the induction hypothesis, $D_{1:(r+\beta)}$ fits $F^{sa,r}_{1:(r+\beta)}$. It is also known that $D_{N\times n}$ passes T. So, as $D_{1:(r+\beta)}$ passes the network consisting of S followed by $RB_{(r+2)\times(r+\beta)}$, $D_{1:(r+1+\beta)}$ passes the network consisting of S followed by $RB_{(r+2)\times(r+1+\beta)}$.

Partition the matrix $D_{1:(r+\beta)}$ into $2^{n-r-\beta}$ submatrices $D^{\gamma}_{2^{r+\beta}\times(r+\beta)}$, $0 \le \gamma \le 2^{n-r-\beta}$, which are labeled in ascending order starting with 0. Let $0 \le u \le 2^{n-r-\beta-1}-1$, $\gamma_1=2u$ and $\gamma_2=2u+1$. The stage $RB_{(r+1+\beta)}$ consists of $2^{n-r-\beta-1}$ copies of the SE stage with $2^{r+1+\beta}$ inputs/outputs. The rows that are sent to the sth, $0 \le s \le 2^{r+\beta}-1$, outputs of the subnetworks that pass $D^{\gamma_1}_{2^{r+\beta}\times(r+\beta)}$ and $D^{\gamma_2}_{2^{r+\beta}\times(r+\beta)}$ enter the sth SB of the SE stage with $2^{r+1+\beta}$ inputs/outputs. Because no conflict occurs in the sth SB by hypothesis, the $(r+1+\beta)$th entries of the rows entering the sth SB must constitute the set $\{0,1\}$. Therefore, by definition of fit, $D_{1:(r+1+\beta)}$ fits $F^{s_2,r}_{1:(r+1+\beta)}$. □

Proof of Corollary VII.1: Consider the network T, that is defined in Theorem VII.1, and its components S and $RB_{(r+2):n}$. Recall that S consists of 2^{n-r-1} copies of a rearrangeable network $P(2^{r+1}!)$ in parallel. If the Benes network $BS_{2^{r+1}\times(2r+1)}$ substitutes for each rearrangeable network $P(2^{r+1}!)$ of S, then S consists of 2^{n-r-1} copies of the rearrangeable network $BS_{2^{r+1}\times(2r+1)}$ in parallel and hence S is made identical to the subnetwork $BS_{(n-r):(n+r)}$ of $BS_{N\times(2n-1)}$. Because $BS_{1:(2n-1)}$ can be considered as being composed of $BE_{1:(n-1)}$ followed by $RB_{1:n}$, $BS_{(n-r):(n+r)}$ is the same as $BE_{(n-r):(n-1)}RB_{1:(r+1)}$. So, T is functionally equivalent to the network consisting of $BS_{(n-r):(n+r)}$ followed by $RB_{(r+2):n}$. Because the network that consists of $BS_{(n-r):(n+r)}$ followed by $RB_{(r+2):n}$ is identical to $BS_{(n-r):(2n-1)}$ and the fact that a balanced matrix $D_{N\times n}$ fits $F^{s_2,r}_{N\times n}$ if and only if $D_{N\times n}$ passes T (Theorem VII.1), $D_{N\times n}$ fits $F^{s_2,r}_{1:n}$ if and only if $D_{N\times n}$ passes $BS_{(n-r):(2n-1)}$. Therefore, the corollary holds. □

REFERENCES

[1] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Comput.*, vol. c-24, pp. 1145-1155, Dec. 1975.

[2] N. Linial and M. Tarsi, "Interpolation between bases and the Shuffle-Exchange network," *Europ. J. Combinatorics*, vol. 10, pp. 29-39, 1989.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, New York, 1979.

[4] L. Lovasz and M. D. Plummer, *Matching Theory*, Annals of Discrete Math., 29, North-Holland, 1986.

[5] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. on Comput.*, vol. c-20, pp. 153-161, Feb. 1971.

[6] D. P. Agrawal, "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Comput.*, vol. c-32, pp. 637-648, July 1983.

[7] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[8] A. Waksman, "A permutation network," *Journal of the ACM*, vol. 15, No. 1, pp. 159-163, Jan. 1968.

[9] C. P. Kruskal and M. Snir, "A unified theory of interconnection network structure," *Theoretical Computer Science*, vol. 48, pp. 75-94, 1986.

[10] K. Y. Lee, "On the rearrangeability of 2(log N)-1 stage permutation networks," *IEEE Trans. Comput.*, vol. c-34, pp. 412-425, May 1985.

[11] C. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. on Comput.*, vol. c-29, pp. 696-702, 1980.

[12] T. Etzion and A. Lempel, "An efficient algorithm for generating linear transformations in a shuffle-exchange network," *SIAM J. Comput.*, Vol. 15, No. 1, pp. 216-221, Feb. 1986.

[13] C. Clos, "A study of non-blocking switching networks," *Bell System Tech. J.*, vol. 32, pp. 406-424, 1953.

[14] T.Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. on Comput.*, vol. c-23, pp. 309-318, March 1974.

[15] K.E. Batcher, "The flip network in STARAN," in *Proc. 1976 Int. Conf. Parallel Processing*, pp. 65-71.

[16] L.R. Goke and G.J. Lipovski, "Banyan networks for partitioning multiprocessor systems," in *Proc. 1st Symp. Comput. Architecture*, Dec. 1973, pp. 175-189.

[17] M.C. Pease III, "The indirect binary n-cube multiprocessor array," *IEEE Trans. on Comput.*, vol. c-26, pp. 458-473, May 1977.

[18] A.Y. Oruc and M.Y. Oruc, "Equivalence relations among interconnection networks," *J. of Parallel and Distributed Computing*, vol. 2, pp. 30-49, 1985.

[19] H. Cam and J.A.B. Fortes, "Permutation routing algorithms of frequently used networks," in preparation.

[20] J.J. Rotman, *The theory of groups: An introduction*, Allyn & Bacon, Rockleigh, N.J., 1965.

[21] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, Lexington Books, 1986.

[22] A. Youssef and B. Arden, "A New Approach to Fast Control of $r^2 \times r^2$ 3-stage Benes Networks of $r \times r$ Crossbar Switches," *1990 Comput. Archit. Conf.*, pp. 50-59.

[23] J. Lenfant, "Parallel permutations of data: A Benes network control algorithm for frequently used permutations," *IEEE Trans. on Comput.*, vol. c-27, pp. 637-647, July 1978.

[24] D. Nassimi and S. Sahni, "A self-routing Benes network and parallel permutation algorithms," *IEEE Trans. on Comput.*, vol. c-30, pp. 332-340, May 1981.

[25] C.S. Raghavendra and R.V. Boppana, "On Self-Routing in Benes and Shuffle-Exchange Networks," *IEEE Trans. on Comput.*, vol. 40, No. 9, pp. 1057-1064, Sept. 1991.

[26] J.C. Bermond, J.M. Fourneau and A. Jean-Marie, "A graph theoretical approach to equivalence of multistage interconnection networks," *Discrete Applied Mathematics*, vol. 22, pp. 201-214, 1989.

[27] D.M. Dias, *Packet Communication in Delta and Related Networks*, Ph.D. Thesis, Rice Univ., 1981.

[28] H. Cam, *Design and permutation routing algorithms of rearrangeable networks*, Ph.D. Thesis, Purdue Univ., May 1992.

[29] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.

REFERENCE NO. 20

Cam, H. and Fortes, J. A. B., "New Routing Algorithms for Benes and Reduced $\Omega_N\Omega_N^{-1}$ Networks," Submitted for publication in IEEE Transactions on Computers.

Note - this paper presents algorithms for the routing of messages in two important rearrangeable networks. They can be used to set processor array systems into configurations that exclude faulty components.

NEW ROUTING ALGORITHMS FOR BENES AND REDUCED $\Omega_N\Omega_N^{-1}$ NETWORKS

Hasan Çam and Jose'A. B. Fortes

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285

ABSTRACT

In this paper, new permutation routing algorithms are presented for the Benes and the reduced $\Omega_N\Omega_N^{-1}$ networks. These algorithms first compute a routing tag of $2n-1$ bits for every input, then set the switches one stage at a time, the setting of a switch in stage j being determined by the jth bit of the routing tags at its inputs. The last n bits of any routing tag are the destination address bits. The serial and parallel time complexities of these algorithms are $O(N\log_2 N)$ and $O(\log_2^2 N)$, respectively. They are the same or better than those of previously proposed routing algorithms. In comparison with other approaches of similar asymptotic complexity, the algorithms in this paper are simpler and easier to understand and program.

Index Terms— Interconnection network, permutations, routing algorithm, frames, balanced matrices, rearrangeability.

This research was supported in part by the Office of Naval Research under contract No. 00014-90-J-1483 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract No. 00014-88-k-0723.

I. Introduction

An interconnection network (IN) is used for exchanging of information among the computation nodes and memory modules in a parallel computer. An IN is often required to realize any permutation between processors and processors/memory modules because many parallel algorithms are designed on the basis of one-to-one type data transfers. If an IN realizes any permutation in one pass, then it is called rearrangeable. A multistage IN of 2×2 switching boxes (SBs) must have at least $2n-1$ stages to be rearrangeable [7]. A disadvantage of this type of networks is the fact that their routing algorithms have costly implementations. This paper presents new simple routing algorithms to realize permutations in these networks. These algorithms, which make use of balanced matrices [2], frames [18] and perfect matching graphs [3,4], take $O(N\log N)$ time on a uniprocessor computer and $O(\log^2 N)$ on a parallel computer of N processors.[1]

Routing algorithms for Benes network, a well-known rearrangeable network [6], are described in [13,14,15,16]. The control algorithm presented in [16], called looping algorithm, determines the settings of the SBs recursively starting with outermost stages and heading towards the center stage. This algorithm takes $O(N\log N)$ time on a uniprocessor. Nassimi and Sahni [15] proposed a parallel algorithm which determines the switch settings in $O(\log^2 N)$ time when a fully interconnected network of N processors is used. Lee [13] introduced a non-recursive algorithm which sets SBs stage by stage from left to right. The advantages of a non-recursive algorithm include a reduction of information transfer among the chips in VLSI Benes network implementations and the pipelining of consecutive permutations so that the routing time is reduced by a factor of $O(\log N)$. Taking into consideration the number of comparisons needed to pair switches such that every consecutive pair has a common residue class number (see [13] and Section III of this paper), the time complexity of the algorithm is $O(N^2\log N)$ and $O(N^2)$ on a uniprocessor and a parallel computer, respectively. The routing algorithm proposed in this paper for Benes network is also non-recursive, but it takes $O(N\log N)$ time on a uniprocessor computer and $O(\log^2 N)$ on a parallel computer. In addition, it is easy to comprehend and simple to program.

Lee [9] has proven the rearrangeability of the network called the reduced $\Omega_N\Omega_N^{-1}$, which is composed of the $(n-1)$-stage SE network followed by the n-stage inverse SE network. The routing algorithm proposed in [9] for reduced $\Omega_N\Omega_N^{-1}$ takes the same time as the one proposed in [13] for the Benes network

[1] All logarithms are in base 2 unless stated otherwise.

discussed in the above paragraph. A new proof for the rearrangeability of this network is also provided in this paper. This proof directly leads to a routing algorithm which is very similar to the routing algorithm of Benes network. This proposed algorithm also takes $O(N\log N)$ time on a uniprocessor computer and $O(\log^2 N)$ on a parallel computer.

The remaining of this paper is organized as follows. Section II is dedicated to the basic terminology and definitions used throughout the paper. Section III presents basic serial and parallel algorithms used in the routing algorithms of the Benes and reduced $\Omega_N\Omega_N^{-1}$. Section IV is dedicated to the routing algorithm of the Benes network. Section V discusses the routing algorithm of the reduced $\Omega_N\Omega_N^{-1}$. Concluding remarks are made in Section VII. The proofs omitted in the paper are presented in the Appendix (Section VI).

II. BASIC DEFINITIONS

Throughout this paper, matrices are denoted by single capital letters and columns of a matrix are represented by the lower case of the capital letter denoting that matrix. Matrix A having N rows and k columns is denoted by $A_{N\times k}$. Given a matrix, e.g. $A_{N\times k}$, the jth column is denoted by a_j, $1 \leq j \leq k$. To be able to refer to a set of specific columns of a matrix, the notation $A_{x:y}$ is used to denote the submatrix that contains those columns of A whose indices are x, $x+1$, $\cdots$,y, where $1 \leq x \leq y$; if x happens to be greater than y, then $A_{x:y}$ refers to a nil matrix, unless stated otherwise. If $x=y$, then $A_{x:y}$ refers to a single column a_x. Unless specifically stated, the number of the rows of a matrix $A_{x:y}$ is assumed to be equal to N. $A_{N\times n}(i)$ refers to the ith row of the matrix $A_{N\times n}$, where $0 \leq i \leq N-1$. A column vector of N entries of which half are 0's and the other half are 1's is called a *column permutation*. Unless otherwise stated, any column of any matrix in this paper is a column permutation. The binary representation of a positive integer $0 \leq b \leq N-1$ is $(b_1 b_2 \cdots b_n)$ such that $b = b_1.2^{n-1} + b_2.2^{n-2} + \cdots + b_n.2^0$.

A *permutation* on a set X is a bijection of X onto itself. A permutation f permutes the ordered list $0, 1, \cdots, N-1$ into $f(0), f(1), \cdots, f(N-1)$. A cyclic notation [19,20] can be used to represent a permutation as the product of cycles, where a cycle $(c_0\ c_1\ c_2\ \cdots\ c_{k-1}\ c_k)$ means $f(c_0) = c_1$, $f(c_1) = c_2$, $\cdots$, $f(c_{k-1}) = c_k$, and $f(c_k) = c_0$. The composition of several permutations $f_1.f_2 \cdots f_k$ is evaluated from left to right, i.e., it maps i into $f_k(\cdots(f_2(f_1(i)))\cdots)$.

Definition II.1. (Permutation matrix, identity permutation matrix, reverse permutation matrix): A permutation h can be

represented by a $N \times n$ binary matrix called *permutation matrix, II*, such that its ith row, $H_{N \times n}(i)$, is the binary representation of the integer $h(i)$. The *identity permutation matrix* denoted by $I_{N \times n}$ is the matrix whose ith row is the binary representation of i (this is called "standard matrix" in [12]). The *reverse permutation matrix,* denoted $R_{N \times n}$, is the matrix whose jth column is the $(n+1-j)$th column of $I_{N \times n}$.

In the terminology used in this paper, a k-stage IN consists of k columns of switching boxes (SBs), each followed and preceded by links which form *interconnection patterns (IPs)* as shown in Figure II.1. The *IPs* formed by the input and output links are denoted by IP_{in} and IP_{out}, respectively. Thus, an IN contains $(k+1)$ interconnection patterns labeled IP_{in}, IP_1, IP_2, . . . , IP_{k-1}, IP_{out}. A column of IN contains $N/2$ (2×2) SBs, each of which can be set either straight or cross. Figures II.2, II.3, and II.4 show several networks considered in this paper for $N=16$, namely, reverse baseline, baseline, Benes, the 4-stage shuffle-exchange (SE), and the 4-stage inverse SE. If some networks are placed in parallel to form a new IN, then the IN is said to be a "pile of networks". Unless otherwise stated, any IN is assumed to have N inputs/outputs and its stages are labeled from left to right starting with 1. Network stages are defined below and illustrated in the figures.

Definition II.2. (Stages of reverse baseline, baseline, Benes, SE, and inverse SE networks): With one exception, a stage in the reverse baseline and SE networks consists of a connection pattern and the following column of SBs. The exception is the rightmost stage (i.e., the output stage) which consists of the last column of SBs and both the preceding and succeeding connection patterns. Stages are labeled from left to right in ascending order starting with 1. In the baseline network the kth stage corresponds to the $(n-k+1)$th stage of the reverse baseline network. (Notice that both the reverse baseline and the baseline can have at most n stages, by definition). In the inverse SE network with m stages, its kth stage corresponds to the $(m-k+1)$th stage of the m-stage SE network. In this paper, Benes network is considered as being composed of the first $n-1$ stages of the n-stage baseline followed by the n-stage reverse baseline. (It could also be considered as being composed of the n-stage baseline followed by the last $n-1$ stages of the n-stage reverse baseline). Therefore, the stages of Benes network are labeled according to the labeling rules of the baseline and the reverse baseline.

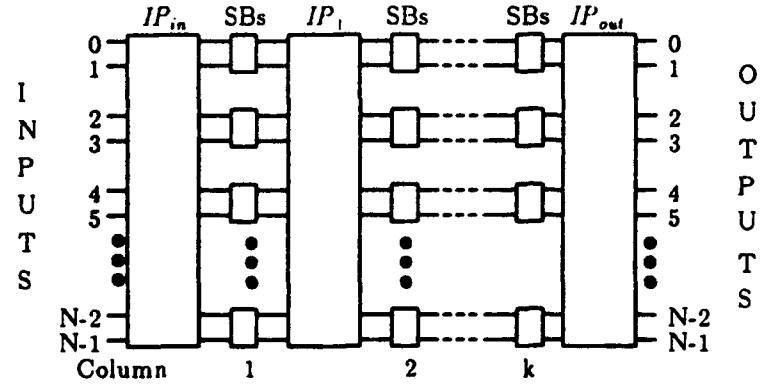


Figure II.1. An IN with (2×2) SBs and interconnection patterns shown as large boxes.

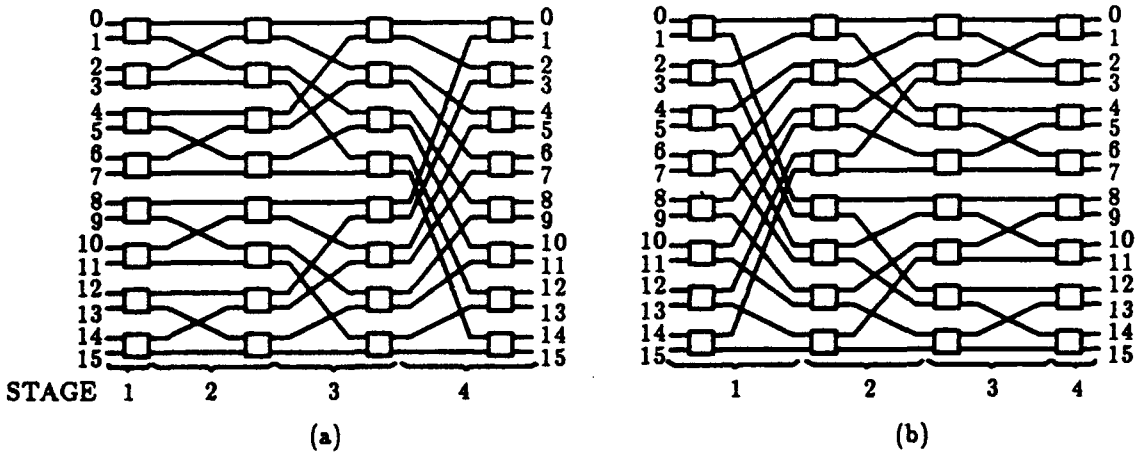


Figure II.2. (a) The 4-stage reverse baseline network with 16 inputs/outputs. (b) The 4-stage baseline network with 16 inputs/outputs.

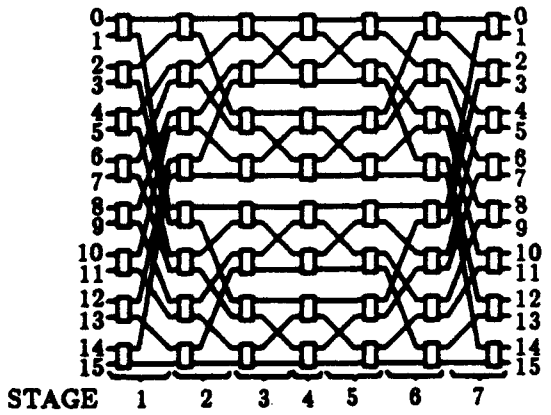


Figure II.3. Benes network with 16 inputs/outputs.

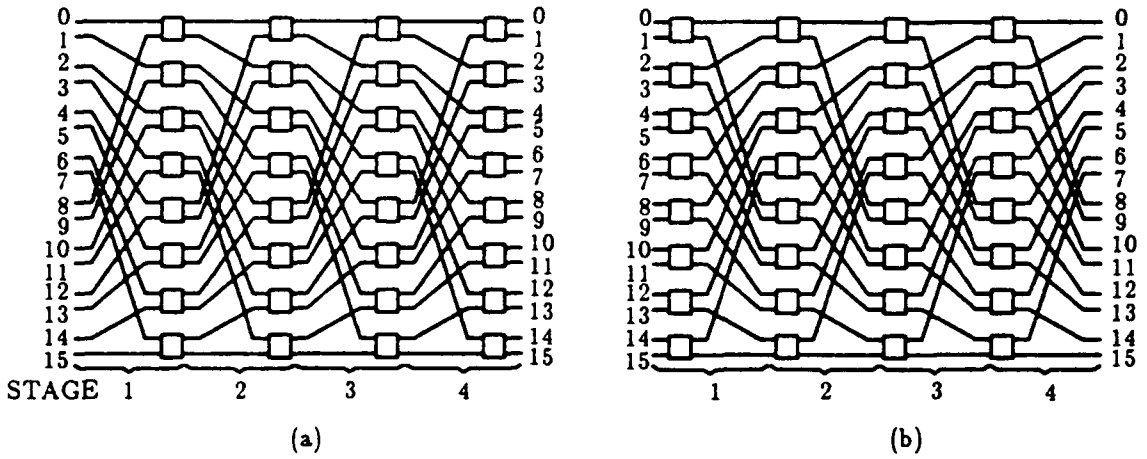


Figure II.4. (a) The omega network (i.e., the 4-stage SE) with 16 inputs/outputs. (b) The inverse omega network (i.e., the 4-stage inverse SE) with 16 inputs/outputs.

An IN with N inputs/outputs and k stages is denoted by both $IN_{N\times k}$ and $IN_{1:k}$, where $k \geq 1$. The subnetwork that consists of the stages x through y of $IN_{1:k}$ is denoted by $IN_{x:y}$, where $1 \leq x<y \leq k$. If $x>y$, then $IN_{x:y}$ refers to a nil network, unless specified otherwise. IN_j, $1 \leq j \leq k$, refers to the jth stage of $IN_{1:k}$. The notation used for networks is different from that used for matrices because matrices are always denoted by single letters.

In this paper the following convention is adopted to denote an IN: if the name of an IN has more than one word, then it is denoted by the upper case form of the first letters of those words; otherwise, it is denoted by the upper case form of its first and last letters. Also, if XX denotes an IN, then the *inverse* XX network may be denoted by XX^{-1}. The following definition applies this convention to the baseline, reverse baseline, shuffle-exchange, inverse shuffle-exchange and Benes networks of interest in this paper.

Definition II.3. (BE, RB, SE, SE^{-1}, BS, composite IN, IP^s): The symbols BE, RB, SE, SE^{-1} and BS in this paper refer to the networks baseline, reverse baseline, shuffle-exchange, inverse shuffle-exchange and Benes, respectively. If an IN is a cascade of different INs, then it is called a *composite IN* and is denoted by the concatenation of symbols that represent the *INs* in the order they are cascaded. IP^s denotes the shuffle interconnection pattern of a SE stage of N inputs/outputs.

As an example for a composite network, the notation $RB_{1:n}SE_{1:m}$, $m \geq 1$, denotes the network consisting of $RB_{1:n}$ followed by $SE_{1:m}$. The reduced $\Omega_N\Omega_N^{-1}$ can also be denoted by both $SE_{1:(n-1)}IP^sSE_{1:n}^{-1}$ and

$SE_{N\times(n-1)}IP^sSE^{-1}_{N\times n}$.

Linial and Tarsi [2] introduced the concept of balanced matrices to establish a relation between SE networks and their realizable permutations. The following definition is equivalent to the one given in [2].

Definition II.4. (Balanced matrix): Let $N=2^n$ and call a 0-1 matrix $A_{N\times k}$ *balanced* if either one of the following conditions is satisfied:

1. For $k \leq n$, it consists of any k columns of the binary representation of a permutation on the set $\{0,1,\ldots,N-1\}$.
2. For $k>n$, every n consecutive columns form the binary representation of a permutation on the set $\{0,1,\ldots,N-1\}$.

As an example, two balanced matrices E and H are shown below. But notice that the matrix $[E\ H]$ is not balanced.

$$E = [e_1\ e_2\ e_3] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \qquad H = [h_1\ h_2] = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Definition II.5. (Pass, realize): A balanced matrix $A_{N\times k}$ (respectively, an *IN*) is said to *pass* a k-stage IN (respectively, a matrix $A_{N\times k}$) if no conflict occurs in the SBs of the IN when $A_{N\times k}(i)$ is used as the routing tag for the *ith* input of the IN. A network *IN realizes* a permutation represented by $B_{N\times n}$ if there is a network switch setting such that input i is sent to output $B(i)$ for all $i=0,1,\ldots,N-1$.

According to the last definition, in this paper, the phrases "an IN passes a balanced matrix" and "a balanced matrix passes an IN" are used alternatively. It is also assumed that only "one pass" is allowed through a network to realize a permutation. Therefore, the phrase "one pass" is omitted in the sequel. To emphasize the distinction between the meaning of the terms "pass" and "realize" as used in this paper, it is important to notice that matrix $A_{N\times k}$ in Definition II.5 does not necessarily correspond to the permutation realized by the network *IN*. Indeed, the ith row of $A_{N\times k}$ is the routing tag for input i and it is only when it equals the destination of input i that $A_{N\times k}$ is the permutation realized by *IN*; these cases will become clear in the remainder of the paper.

In the rest of this section, the concept of frames is introduced to characterize the permutations realized by a network. Different frames are derived from this concept and their graphical representations are presented. In addition, some related fundamental concepts used in the proofs of this paper are introduced. More extended discussion of these concepts appears in [17].

In order to understand the concept of frame, the following definition is first introduced (a *k-tuple* V with the elements $v_1, v_2, \ldots, v_k$, denoted by $V = <v_1, v_2, \ldots, v_k>$, refers to an ordered collection of k elements).

Definition II.6. (Partition P_i, block, standard partition P_i^*, P^*): Let $X=\{0, 1, \ldots, N-1\}$, $N=2^n$ and $i=1,2, \ldots, n$. A *partition* P_i of X is a tuple of 2^{n-i} disjoint ordered subsets of X, called *blocks*, each of which is a tuple with 2^i distinct elements. The partition $P_i^*=<<h, h+1, \ldots, h+2^i-1>$ such that $h \ mod \ 2^i = 0$ and $h = 0,1, \ldots, N-1>$ is a *standard partition* of X. The $n-tuple$ $<P_i^*, i=1,2, \ldots, n>$ is denoted by P^*.

Example II.1. Let $N=8$. The following are the standard partitions: $P_1^*=<<0,1>, <2,3>, <4,5>, <6,7>>$, $P_2^*=<<0,1,2,3>, <4,5,6,7>>$ and $P_3^*=<0,1,2,3,4,5,6,7>$. Also, $P^*=<P_1^*, P_2^*, P_3^*>$.

The notion of frame is defined next and an example (Example II.2) is given after the definition. Several frames are graphically represented in Figure II.5. Notice that the frames are characterized by the labeling of columns, the labeling of rows, and the number of blocks in columns. Therefore, the definition of frame is done in terms of two mappings (the column and row labeling) and a tuple of partitions (one for each column). The column labels determine the number and size of the blocks in each partition and the row labeling determines the elements in each block and their order. As precisely stated in the definition, column with label $\beta(i)$ corresponds to a partition with $2^{n-\beta(i)}$ blocks with $2^{\beta(i)}$ elements each and the mth element within the jth block corresponds to the label $\gamma(r)$ of row $r = 2^{\beta(i)}(j-1)+m-1)$. After Example II.2, a convenient graphical representation for frames is introduced and its use is illustrated in Example II.3 for the frames described in Example II.2.

Definition II.7. (Frame): Let $1 \leq k \leq n$ and $1 \leq i \leq k$. A *frame* $F_{N\times k}$, $1 \leq k \leq n$, is a 3-tuple $<\beta, \gamma, P>$, where

- β is a mapping of the set $\{1,2, \ldots, k\}$ into $\{1,2, \ldots, n\}$,
- γ is a permutation on the set $\{0,1, \ldots, N-1\}$ and
- P is a tuple of partitions $<P_{\beta(1)}, P_{\beta(2)}, \ldots, P_{\beta(k)}>$ determined by β and γ as follows:

$P_{\beta(i)} = <P_{\beta(i),1}, P_{\beta(i),2}, \ldots, P_{\beta(i),2^{n-\beta(i)}}>$ where
$P_{\beta(i),j} = <u_{1,j}, u_{2,j}, \ldots, u_{2^{\beta(i)},j}>$ such that
$u_{m,j} = \gamma(2^{\beta(i)}(j-1)+m-1)$, $1 \leq j \leq 2^{n-\beta(i)}$ and $1 \leq m \leq 2^{\beta(i)}$.

Definition II.8. (a-frame, standard a-frame, b-frame, standard b-frame): Consider the 3-tuple $<\beta,\gamma,P>$ that defines a frame $F_{N\times k}$. If β is the identity permutation, then $F_{N\times k}$ is an *a−frame* denoted by $F^{a}_{N\times k}$. If β and γ are the identity permutations (which implies $P=P^{*}$), then $F_{N\times k}$ is the *standard a-frame* denoted by $F^{sa}_{N\times k}$.

By definition of standard *a*-type frame, column f^{sa}_i, $1 \leq i \leq n$, has 2^{n-i} blocks, each having 2^i rows. Unless otherwise stated, the number of the rows of $F^{sa}_{1:k}$, $k \geq 1$, is assumed to be N. The notation $F_{x:y}$ is used to denote the sub-frame that contains those columns of F whose indices are $x, x+1, \ldots, y$. Unless specifically stated, the number of rows of $F_{x:y}$ is assumed to be N.

Example II.2. The following are examples of frames for $N=8$ and $k=3$.
(*a*) $F_{8\times3}=<\beta,\gamma,P>$ where $\beta = (1\ 2)(3)$, γ is the identity permutation and $P=<P_2,P_1,P_3>$ such that $P_1=P_1^{*}$, $P_2=P_2^{*}$ and $P_3=P_3^{*}$.
(*b*) $F^{a}_{8\times3}=<\beta,\gamma,P>$ where $\beta =$ identity permutation, $\gamma = (0)(1\ 2)(3)(4)(5)(6)(7)$, $P=<P_1,P_2,P_3>$, $P_1=<<0,2>, <1,3>, <4,5>, <6,7>>$, $P_2=<<0,2,1,3>, <4,5,6,7>>$ and $P_3=<0,2,1,3,4,5,6,7>$.
(*c*) $F^{a}_{8\times3}=<\beta,\gamma,P>$ where $\beta =$ identity permutation, $\gamma = (0)(1\ 3\ 6\ 4)(2\ 5)(7)$, $P=<P_1,P_2,P_3>$, $P_1=<<0,3>, <5,6>, <1,2>, <4,7>>$, $P_2=<<0,3,5,6>, <1,2,4,7>>$ and $P_3=<0,3,5,6,1,2,4,7>$.
(*d*) $F_{8\times3}=<\beta,\gamma,P>$ where $\beta = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 3 \end{pmatrix}$, γ is the identity permutation, $P=<P_2,P_2,P_3>$, $P_2=P_2^{*}$ and $P_3=P_3^{*}$.

Definition II.9. (Graphical representation of a frame, rectangle of a frame): The *graphical representation of a frame* $F_{N\times k}=<\beta,\gamma,P>$ consists of k columns labeled f_i, $i=1, \ldots, k$, from left to right and N rows labeled $\gamma(j)$, $j= 0,1, \ldots, N-1$ starting at the top. The column f_i corresponds to the partition $P_{\beta(i)}$, that is, f_i consists of $2^{n-\beta(i)}$ blocks of $2^{\beta(i)}$ entries each. In the graphical representation of a frame, any polygon with four sides and four right angles is a *rectangle of the frame.*

Example II.3. Figures III.1a, III.1b, III.1c and III.1d show the graphical representation of the frames described in the part (*a*), (*b*), (*c*) and (*d*) of Example II.2, respectively. Figure II.5e shows the graphical representation of the standard *a*-frame $F^{sa}_{8\times3}$. The labels of the partitions below each column are implicit by the sizes of the rectangles in the column and can be omitted.

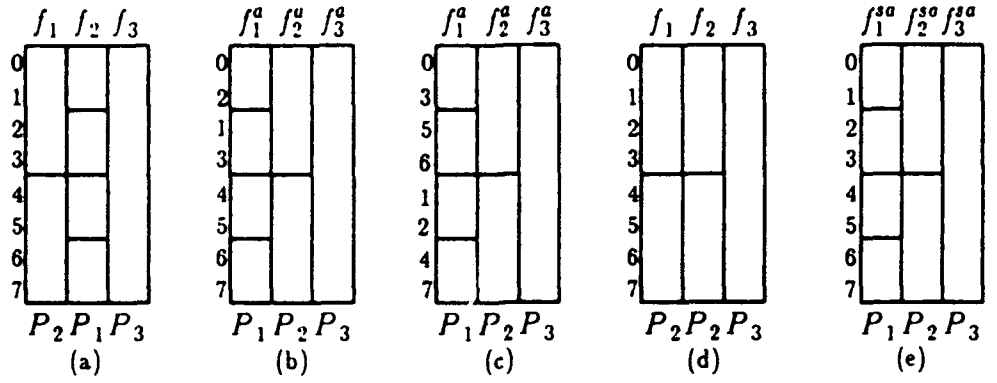


Figure II.5. (a), (b), (c) and (d) are the graphical representations of the frames described in the part (a), (b), (c) and (d) of Example II.2, respectively. (e) Graphical representation of the standard a-frame $F^{sa}_{8\times 3}$.

Definition II.10. (Fit): Let $k \geq 1$, $0 \leq i \leq N-1$ and $1 \leq j \leq k$. Consider a balanced matrix $A_{N\times k}$ and a frame $F_{N\times k}$. The matrix A *fits* $F_{N\times k}$ if and only if, after placing a_{ij} in the ith row and jth column of $F_{N\times k}$, every rectangle of $F_{N\times k}$ contains a balanced matrix.

Example II.4. The matrix E, shown just after Definition II.4, fits all the frames shown in Figure II.5 (a)-(d). It does not fit $F^{sa}_{8\times 3}$ shown in Figure II.5e because, for example, the submatrix in the top leftmost rectangle (the 2-tuple $P_{1,1}$) is not balanced.

III. BASIC ALGORITHMS

This section introduces sequential and parallel algorithms for determining a column vector x for given balanced matrices $A_{N\times(n-1)}$ and $B_{N\times(n-1)}$ such that $[A_{N\times(n-1)}\ x]$ and $[B_{N\times(n-1)}\ x]$ are balanced. These algorithms are later used in obtaining the routing algorithms for the Benes and $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ networks. Some preliminary results and basic definitions are introduced first.

Lemma III.1. [2] For $n \geq 2$, let A and B be two $N\times(n-1)$ balanced matrices. Then there exists a column vector x such that both $[A\ x]$ and $[x\ B]$ are balanced matrices.

Note that, when the order of columns in a balanced matrix with at most n columns is changed, the matrix remains balanced. Therefore, the position of x in the matrices A and B in Lemma III.1 is immaterial. Because the possible choices of vector x increase as the number of columns of A or B is reduced, Lemma III.1 remains valid when A and B have less than $n-1$ columns.

Some properties of balanced matrices can be captured by graphs. A graph $G=(V,E)$ consists of a set of vertices V and a set of edges E, each of which is a pair of vertices. The *union of two graphs* $G_1=(V,E_1)$ and $G_2=(V,E_2)$ is the graph $G=G_1 \cup G_2=(V,E_1 \cup E_2)$. In other words, an edge is present in $G=G_1 \cup G_2$ if and only if it is present in either G_1 or G_2. A subset M of edges in a graph G is called independent or a *matching* if no two edges of M have a vertex in common. A matching M is said to be a *perfect matching* if it covers all vertices of G. More extended discussion of these basic concepts can be found in [3,4].

Definition III.1. (Perfect matching graph of a matrix): Let A be an $N \times k$ $(1 \leq k \leq n-1,\ n \geq 2)$ balanced matrix. A *perfect matching graph of* A, denoted by PG_A, is a graph whose vertices are in one-to-one correspondence with the rows of A, have degree one and vertices v_i and v_j are joined by an edge only if the ith row and jth row of A are identical.

If the number of columns in a balanced matrix $A_{N \times k}$ is less than $n-1$ (i.e., if $k<n-1$), then its perfect matching graph is not unique because each distinct row in A appears 2^{n-k} times. If $k=n-1$, then PG_A is unique because each distinct row in A appears twice. As an example, consider the balanced matrix $H_{8\times 2}$ presented just after Definition II.4. Its perfect matching graph is unique and shown in Figure III.1.

Definition III.2. (Labeling): 2*−labeling* or 2*−coloring* of a graph is the assignment of integers 0 and 1 to its vertices such that the labels of the vertices incident with an edge are different.

Fact III.1. [2] The union of two perfect matching graphs with the same set of vertices is a union of disjoint even cycles and, therefore, it can be 2-labeled.

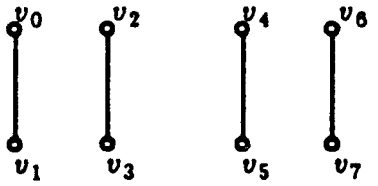


Figure III.1. The perfect matching graph of $H_{8\times 2}$

Given balanced matrices $D_{1:n}$ and $E_{1:n}$, the following algorithm CONS_COLUMN determines a column vector V such that $S_{1:n} = [D_{2:n}\ V]$ and $T_{1:n} = [E_{2:n}\ V]$ are balanced. The detailed steps of this serial algorithm are shown in procedure COLUMN.

Algorithm CONS_COLUMN

Input: Balanced matrices $D_{1:n}$ and $E_{1:n}$.

Output: A vector V such that $[D_{2:n}\ V]$ and $[E_{2:n}\ V]$ are balanced.

Step 1. Determine $D_{1:n}^{-1}$ and $E_{1:n}^{-1}$ which represent the inverse of the permutations corresponding to $D_{1:n}$ and $E_{1:n}$, respectively.

Step 2. Determine the perfect matching graph of $D_{2:n}$ by joining the vertices with indices $D_{1:n}^{-1}(j)$ and $D_{1:n}^{-1}(j+(N/2))$ by an edge, $0 \leq j \leq (N/2)-1$. Similarly, determine the perfect matching graph of $E_{2:n}$ by joining the vertices with indices $E_{1:n}^{-1}(j)$ and $E_{1:n}^{-1}(j+(N/2))$ by an edge.

Step 3. 2-label each cycle of the union of the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$. The vector V is such that $V(r)$ equals the integer assigned to the vertex with index r in 2-labeling, where $0 \leq r \leq N-1$. **Stop.**

To prove the correctness of Algorithm CONS_COLUMN, note that $D_{1:n}^{-1}(i)$ can be thought of as a pointer to the row with value i of $D_{1:n}$ because it equals the index of the row of $D_{1:n}$ whose contents equal i. This implies that any two rows of $D_{1:n}$ whose indices equal $D_{1:n}^{-1}(j)$ and $D_{1:n}^{-1}(j+(N/2))$ differ only in their leftmost bits. When the perfect matching graph of $D_{2:n}$ needs to be constructed, it follows from Definition III.1 that the vertices with indices $D_{1:n}^{-1}(j)$ and $D_{1:n}^{-1}(j+(N/2))$ must be joined by an edge. Similarly, in case of the perfect matching graph of $E_{2:n}$, the vertices with indices $E_{1:n}^{-1}(j)$ and $E_{1:n}^{-1}(j+(N/2))$ must be joined by an edge. By Fact III.1, the union of two perfect matching graphs can be labeled in the way that the labels of the vertices incident with an edge constitute the set $\{0,1\}$. This completes the correctness proof of Algorithm CONS_COLUMN.

Procedure COLUMN, an implementation of Algorithm CONS_COLUMN, is now discussed in order to rigorously analyze the complexity of the algorithm. It calls two simple procedures namely CONS_LIST(N) and DELETE(k) which constructs a linear linked list of N nodes and deletes the kth node of the list, respectively (these procedures are described in Appendix). Given balanced matrices $D_{1:n}$ and $E_{1:n}$, procedure COLUMN determines a column vector V such that the matrices $[D_{2:n}\ V]$ and $[E_{2:n}\ V]$ are balanced. The lines 1 to 4 in procedure COLUMN construct $D_{1:n}^{-1}$ and $E_{1:n}^{-1}$ which represent the inverse of the permutation represented by $D_{1:n}$ and $E_{1:n}$, respectively. The lines 5 to 10 create as many circular doubly linked lists as the number of cycles in the union of the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$. (Recall that any of these cycles has an even number of vertices). Any node of these lists has three fields called R_1, R_2 and V, where R_1 and R_2 are used for pointers and V will, later,

be assigned either 0 or 1 in 2-labeling. The fields R_1 (respectively, R_2) are used to obtain the perfect matching graph of $D_{2:n}$ (respectively, $E_{2:n}$). The pointers R_1 and R_2 of any one of these lists are ordered as $(R_1R_2R_2R_1 \cdots R_1R_2R_2R_1)$. The line 11 calls the procedure CONS_LIST(N) to construct a linear linked list, called H, of N nodes. The lines 12 to 24 do the 2-labeling of all the cycles in the union of the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$ by assigning 0 or 1 appropriately to the fields V of nodes of the corresponding circular doubly linked lists. The 2-labeling of the cycle containing the vertex of index 0 is done first. Whenever the field V of a node is assigned either 0 or 1, the node is deleted from the list H, so that H contains only those nodes which are not assigned any integer yet. If the node to be deleted is the header node, the pointer to the header is changed to point to the node following the header node. When all the nodes of H are deleted, it means that all the cycles are 2-labeled. When the 2-labeling of a cycle is completed (i.e., when the "while" loop between the lines 18 and 23 is exited), the next cycle to be 2-labeled is the one that contains the node with the index p pointing to the header of H. So, the lines 12 to 24 do the 2-labeling of N nodes one by one. Also, note that the "for" loops are executed $O(N)$ times. Therefore, the time complexity of procedure COLUMN is $O(N)$.

```
line  procedure COLUMN
1         for i := 0 to N−1 do
2             D^{-1}_{1:n}(D_{1:n}(i)) := i
3             E^{-1}_{1:n}(E_{1:n}(i)) := i
4         end

5         for j := 0 to (N/2)−1 do
6             R_1(D^{-1}_{1:n}(j)) := D^{-1}_{1:n}(j+(N/2))
7             R_1(D^{-1}_{1:n}(j+(N/2))) := D^{-1}_{1:n}(j)
8             R_2(E^{-1}_{1:n}(j)) := E^{-1}_{1:n}(j+(N/2))
9             R_2(E^{-1}_{1:n}(j+(N/2))) := E^{-1}_{1:n}(j)
10        end

11        call CONS_LIST(N)
12        p := 0
13        while p ≠ nil do
14            V(p) := 0; V(R_1(p)) := 1
15            m := R_2(R_1(p))
16            call DELETE(p)
17            call DELETE(R_1(p))
```

```
18          while V(m) is empty do
19              V(m) := 0; V(R1(m)) := 1
20              m := R2(R1(m))
21              call DELETE(m)
22              call DELETE(R1(m))
23          end
24      end
25  end COLUMN
```

Algorithm CONS_COLUMN and procedure COLUMN can be adapted for execution on an N processor PRAM machine [21], as shown next in Algorithm CONS_COLUMN_PRAM and procedure COLUMN_PRAM.

Algorithm CONS_COLUMN_PRAM

Input: Balanced matrices $D_{1:n}$ and $E_{1:n}$.

Output: A column vector V such that $[D_{2:n}\ V]$ and $[E_{2:n}\ V]$ are balanced.

Step 1. Determine the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$.

Step 2. Represent each cycle of the union of the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$ by a circular doubly linked list.

Step 3. For each list, first determine the node with smallest index and then assign "nil" to one of its pointers.

Step 4. For each list, determine the distance of each node to the node with smallest index.
/* Note that, although the distance of any node of a circular doubly linked list to the node of smallest index can be measured in two different directions, both distances of the same node are either odd or even because the list contains an even number of nodes. */

Step 5. For each list, assign 0 (respectively, 1) to all those nodes whose distances are even (respectively, odd). **Stop.**

The correctness proof of Algorithm CONS_COLUMN_PRAM is the same as that of Algorithm CONS_COLUMN, except that Algorithm CONS_COLUMN_PRAM is implemented using N processors instead of a single processor.

Given balanced matrices $D_{1:n}$ and $E_{1:n}$, procedure COLUMN_PRAM determines a column vector V using an N processor PRAM model such that the matrices $[D_{2:n}\ V]$ and $[E_{2:n}\ V]$ are balanced. Assignments which are local and not local to the processors are denoted by the operators ":=" and "←", respectively. Any variable with index x is an operand in processor $PE(x)$, where

$0 \leq x \leq N-1$. Lines 1 and 2 determine the inverse of the permutations represented by $D_{1:n}$ and $E_{1:n}$. Lines 3 to 10 represent each cycle of the union of the perfect matching graphs of $D_{2:n}$ and $E_{2:n}$ by a circular doubly linked list. The pointers R_1 and R_2 of any one of these lists are ordered as $(R_1R_2R_2R_1 \cdots R_1R_2R_2R_1)$. Let $2 \leq p \leq n$. Any one of these lists can have at most 2^k nodes, where $k=p-1$ if $D_{p:n} = E_{p:n}$, else $k=n$.

Lines 11 and 12 copy the circular lists into new lists whose pointers are denoted by S_1 and S_2. Lines 13 to 21 determine the node of the smallest index in each list and the smallest index is stored in $M(i)$. Line 13 initializes $M(i)$ to i. Lines 14 and 15 find the smaller index of every two nodes that point to each other by S_1 fields and then the $M(i)$s of these nodes are assigned the smaller index. Line 16 partitions each circular linked list into two sublists, such that each sublist is formed by the S_1 fields of the nodes and contains every other node of the list. Due to the lines 14 and 15, the smallest one of the node indices of a list is stored in the $M(i)$ of a node in the every sublist. This guarantees that the $M(i)$s of all the nodes of a list will eventually contain the same index even though the list is partitioned into two sublists. Because a sublist can have at most 2^{k-1} nodes and, after the jth iteration, $S_1(i)$ of node i points to the node which is 2^j nodes away from node i, the loop "for" in lines 17 to 21 needs to be executed at most $k-1$ times. Hence, the loop "for" takes $O(k)$ times.

Lines 22 to 35 determine $d(i)$ which is the distance of node i to the node of the smallest index. Line 23 partitions each linked list into two sublists, each formed by the R_1 fields of the nodes and contains every other node of the original list. The distance $d(i)$ is first initialized to 2 (line 24), then $d(i)$s of the node with the smallest index and its neighboring nodes are made equal to 0, 1, and 1, respectively, and their $R_1(i)$s are assigned "nil" (lines 25 to 29). The reason why $d(i)$ of all the nodes other than the nodes with the smallest index and its neighboring nodes are equal to 2 initially is that every sublist of a list contains every other node of a list. For the same reason, one of the two sublists contains the node with the smallest index while the other sublist contains the neighboring nodes of the node with the smallest index. So, a sublist has at least one node with R_1 field assigned "nil". The distances of the other nodes of every list to the node of the smallest index are computed in the loop "while" between the lines 30 to 35. This loop can be executed at most k times because, after the jth iteration, $R_1(i)$ of node i of a sublist points to the node which is 2^j nodes away from node i, and $R_1(i)$ is assigned "nil" when the R_1 field of the node pointed to by the $R_1(i)$ equals "nil". Lines 36 and 37 do the 2-labeling of the nodes of every list. Because the loops "for" and "while" of procedure COLUMN_PRAM take $O(\log N)$ times, its time complexity equals $O(\log N)$.

```
line  procedure COLUMN_PRAM
1         D^{-1}_{1:n}(D_{1:n}(i)) ← i
2         E^{-1}_{1:n}(E_{1:n}(i)) ← i
3         if i ≥ 0 and i ≤ (N/2)−1 then do
4                         R_1(D^{-1}_{1:n}(i)) ← D^{-1}_{1:n}(i+(N/2))
5                         R_2(E^{-1}_{1:n}(i)) ← E^{-1}_{1:n}(i+(N/2))
6                                              end
7         if i ≥ N/2 and i ≤ N−1 then do
8                         R_1(D^{-1}_{1:n}(i)) ← D^{-1}_{1:n}(i−(N/2))
9                         R_2(E^{-1}_{1:n}(i)) ← E^{-1}_{1:n}(i−(N/2))
10                                             end
11        S_1(i) := R_1(i)
12        S_2(i) := R_2(i)
13        M(i) := i
14        T(i) ← M(S_1(i))
15        M(i) := min{T(i),M(i)}
16        S_1(i) ← S_2(S_1(i))
17        for j := 1 to k-1 do
18            T(i) ← M(S_1(i))
19            S_1(i) ← S_1(S_1(i))
20            M(i) := min{T(i),M(i)}
21        end
22        t(i) := R_1(i)
23        R_1(i) ← R_2(R_1(i))
24        d(i) := 2
25        if M(i) = i then do
26                        d(i) := 0 ; R_1(i) := nil
27                        d(t(i)) ← 1 ; R_1(t(i)) ← nil
28                        d(R_2(i)) ← 1 ; R_1(R_2(i)) ← nil
29                                 end
30        while there exists a node i such that R_1(i) ≠ nil do
31              if R_1(i) ≠ nil then do
32                        d(i) ← d(i) + d(R_1(i))
33                        R_1(i) ← R_1(R_1(i))
34                                        end
35        end
36        if d(i) = 2×⌊d(i)/2⌋ then V(i) := 0
37                             else V(i) := 1
38    end COLUMN_PRAM
```

An example is presented below to show how circular doubly linked lists are used to represent the cycles of the union of the perfect matching graphs of two given balanced matrices. Also, the 2-labeling of these lists and the resulting column vector are shown.

Example III.1. Let $\Pi = (0\ 4\ 1\ 10\ 9\ 6\ 15\ 3\ 5\ 7)(2\ 11)(8\ 12\ 13)(14)$ and $\Phi = (0\ 2)(1\ 3\ 9\ 4)(5\ 8\ 10\ 11)(6\ 12\ 7\ 13\ 15\ 14)$ be the given permutations. Assume that $D_{1:4}$ and $E_{1:4}$ represent Π and Φ in binary, respectively. The inverse of these permutations, denoted by Π^{-1} and Φ^{-1}, can be easily determined: $\sigma = \Pi^{-1} = (0\ 7\ 5\ 3\ 15\ 6\ 9\ 10\ 1\ 4)(2\ 11)(8\ 13\ 12)(14)$ and $\mu = \Phi^{-1} = (0\ 2)(1\ 4\ 9\ 3)(5\ 11\ 10\ 8)(6\ 14\ 15\ 13\ 7\ 12)$. The perfect matching graphs of $D_{2:4}$ and $E_{2:4}$, denoted by PG_D and PG_E, respectively, are illustrated in Figure III.2. The union of the perfect matching graphs of $D_{2:4}$ and $E_{2:4}$, denoted by $PG_{D\cup E}$, is illustrated in Figure III.3.

Each cycle of $PG_{D\cup E}$ is represented by a circular doubly linked list such that the pointer field R_1 (respectively, R_2) of each node is used for PG_D (respectively, PG_E). The corresponding lists are shown in Figure III.4a. Each list (as well as a cycle) can be 2-labeled in two different ways. The integers assigned to the nodes of a list in a 2-labeling are stored in their V fields. The column vector V obtained from these 2-labelings is shown with the given matrices $D_{1:4}$ and $E_{1:4}$ in Figure III.4b.

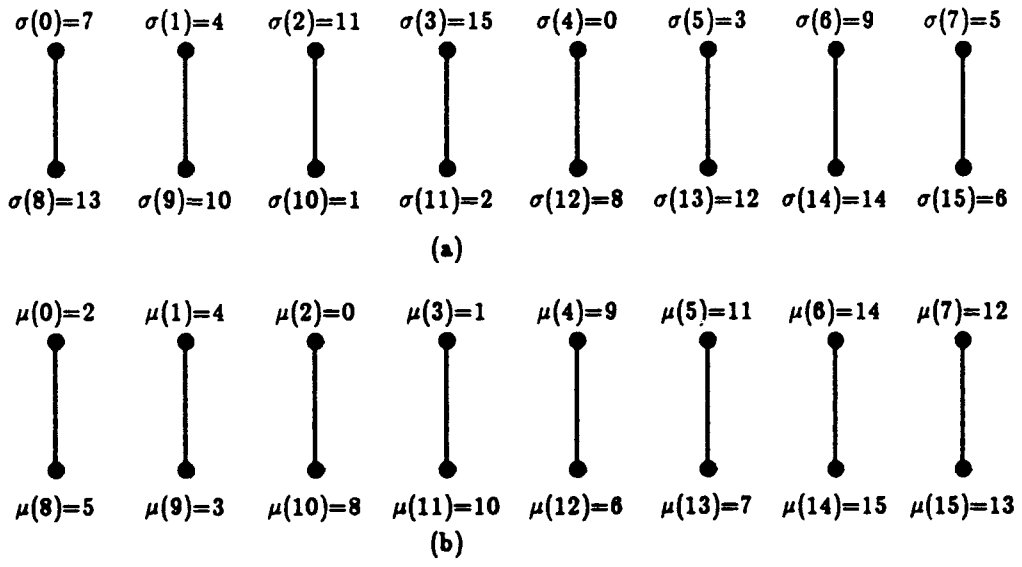


Figure III.2. **(a) PG_D, the perfect matching graph of $D_{2:4}$. (b) PG_E, the perfect matching graph of $E_{2:4}$. The numbers on an edge refer to the indices of its vertices.**

Although this algorithm works like Lee's algorithm in the sense that it sets the SBs of a stage at a time, starting with the leftmost stage and heading towards the rightmost stage, it is easy to understand and simple to program.

When the Benes network realizes a permutation represented by a balanced matrix $B_{1:n}$, the mth input of the Benes network is sent to the output whose value equals $B_{1:n}(m)$, $0 \leq m \leq N-1$. In order for a balanced matrix $B_{1:n}$ to also pass the Benes network, $B_{1:n}$ must be transformed by $BE_{1:(n-1)}$ into another balanced matrix which can pass $RB_{1:n}$. Algorithm CNTR_BENES first determines a balanced matrix $C_{1:(n-1)}$ which passes $BE_{1:(n-1)}$ and transforms $B_{1:n}$ into another balanced matrix passing $RB_{1:n}$. Algorithm CNTR_BENES, then, determines the routing tag matrix $U_{1:(2n-1)}$ whose mth row is the routing tag for the mth input of Benes network to implement a given permutation matrix $B_{1:n}$ through it.

The routing scheme used in Benes network is as follows: at the jth stage, $1 \leq j \leq 2n-1$, of Benes network, every SB examines the bit u_j of the routing tag $(u_1 u_2 \cdots u_{2n-1})$ of its each input. If u_j=0, the input is sent to the upper output of the SB; otherwise, the lower output of the SB is taken.

Algorithm CNTR_BENES

Input: A balanced matrix $B_{1:n} = [b_1\ b_2 \cdots b_n]$ and the reverse permutation matrix $R_{1:n}$.

Output: A matrix $U_{1:(2n-1)}$ such that, when its mth row, $0 \leq m \leq N-1$, is used as the routing tag for the mth input of the Benes network, the permutation represented by $B_{1:n}$ is realized by the Benes network.

Step 1. Let p denote an integer variable. Let $B^r_{1:n} = [b_n\ b_{n-1} \cdots b_1]$. Set p=1. Determine a column vector c_1 such that the matrices $[R_{2:n}\ c_1]$ and $[B_{1:(n-1)}\ c_1]$ are balanced.

Step 2. Increment p by 1. If $p>n-1$, go to next step; otherwise, first determine a column vector c_p such that $[R_{(p+1):n}\ C_{1:(p-1)}]$ and $[B^r_{1:(n-p)}\ C_{1:(p-1)}]$ are balanced, and then go to Step 2.

Step 3. Let $U_{1:(2n-1)} = [C_{1:(n-1)}\ B_{1:n}]$. A switch at the jth stage, $1 \leq j \leq 2n-1$, of the Benes network examines the bit u_j of the routing tag $(u_1\ u_2 \cdots u_{2n-1})$ of its input to set itself either straight or cross. **Stop.**

Theorems IV.1 and IV.2 below are used in the correctness proof of Algorithm CNTR_BENES. The proof of Theorem IV.2 appears in Appendix. Theorem IV.1 establishes the correspondence between $RB_{1:k}$ and $F^{sa}_{1:k}$.

Theorem IV.1. A matrix $D_{1:k} = [d_1\ d_2\ \cdots\ d_k]$ fits $F^{sa}_{1:k}$ if and only if $D_{1:k}$ passes $RB_{1:k}$, $1 \leq k \leq n$. Moreover, $RB_{1:k}$ sends its ith input to its jth output, where j is equal to the sum of $(\lfloor i/2^k \rfloor \times 2^k)$ and the value of $D_{1:k}(i)$.

Basic idea of proof:

($\longrightarrow$) $D_{1:k}$ fits $F^{sa}_{1:k} \longrightarrow D_{1:k}$ passes $RB_{1:k}$.

Induction on k is used. For k=1, each rectangle of $F^{sa}_{1:k}$ has a 0 and a 1. These are the control bits of a switch in $RB_{1:k}$ and, thus, no conflict occurs. For k>1, assuming the theorem holds for $k-1$, each switch in the kth stage "has" control bits 0 and 1 and, therefore, no conflicts occur. These control bits must appear as the kth bits at the end of identical $(k-1)$-bit rows of subframes $F^{o}_{2^{i-1}\times k-1}$ and $F^{e}_{2^{i-1}\times k-1}$ of $F^{sa}_{1:k}$ so that $D_{1:k}$ fits $F^{sa}_{1:k}$. Each subframe corresponds to a subnetwork of $RB_{1:k}$ which is also a reverse baseline network $RB_{2^{i-1}\times k-1}$.

($\longleftarrow$) $D_{1:k}$ passes $RB_{1:k} \longrightarrow D_{1:k}$ fits $F^{sa}_{1:k}$.

Induction on k is used. For k=1, if d_1 passes RB_1, then each rectangle of $F^{sa}_{1:k}$ contains a 0 and a 1 and d_1 fits f^{sa}_1. For k>1, assuming the theorem holds for $k-1$, for the outputs of two subnetworks $RB^{o}_{2^{i-1}\times k-1}$ and $RB^{e}_{2^{i-1}\times k-1}$ to cause no conflict in any switch of the kth stage it must be the case that a 0 and a 1 are added to the $k-1$ entries of identical rows of the frames that correspond to the two subnetworks. This implies that $D_{1:k}$ fits $F^{sa}_{1:k}$. The value of j follows from the topology of $RB_{1:k}$ and how switches are set by control bits. □

Theorem IV.2. Consider a balanced matrix $D_{N\times k}$, $1 \leq k \leq n$, the reverse permutation matrix $R_{N\times n}$ and the frame $F^{sa}_{N\times k}$. The matrix $[R_{N\times n}\ D_{N\times k}]$ is balanced if and only if $D_{N\times k}$ fits $F^{sa}_{N\times k}$.

To prove the correctness of Algorithm CNTR_BENES, note that it follows from Algorithms CONS_COLUMN and Algorithm CONS_COLUMN_PRAM that Step 1 and Step 2 of Algorithm CNTR_BENES are realizable. Now, it remains to show that, when the mth row of $U_{1:(2n-1)}$ is used as the routing tag for the mth input of the Benes network, the permutation represented by $B_{1:n}$ is realized on the Benes network.

Because $[R_{1:n}\ C_{1:(n-1)}]$ is balanced, it follows from Theorems IV.1 and IV.2 that $C_{1:(n-1)}$ passes $RB_{1:(n-1)}$. Because $RB_{1:n}$ is functionally and topologically equivalent to $BE_{1:n}$, $RB_{1:(n-1)}$ can be converted to $BE_{1:(n-1)}$ by repositioning its switches only. This implies that $C_{1:(n-1)}$ also passes $BE_{1:(n-1)}$, that is, no conflict occurs in the switches of $BE_{1:(n-1)}$ when the ith input of $C_{1:(n-1)}$ is used as the routing tag for the ith input. Let x denote a column vector such that $[C_{1:(n-1)}\ x]$ is balanced. Because $BE_{1:n}$ (and $RB_{1:n}$) realizes the permutation represented by $[C_{1:(n-1)}\ x]$ (Theorem IV.1), the routing tags form the

matrix $I_{1:n}$ when they reach the outputs of $BE_{1:n}$. This implies that, when the mth row of $C_{1:(n-1)}$ is used as the routing tag for the mth input of $BE_{1:(n-1)}$, the routing tags form the matrix $I_{1:(n-1)}$ at the outputs of $BE_{1:(n-1)}$. So, when the mth row of $U_{1:(2n-1)}$ is used as the routing tag for the mth input of $BE_{1:(n-1)}$, the routing tags form the matrix $U'_{1:(2n-1)} = [I_{1:(n-1)}\ B'_{1:n}]$ at the outputs, where $B'_{1:n}$ denote the matrix obtained from $B_{1:n}$ by routing it with $C_{1:(n-1)}$. It will be shown next that $B'_{1:n}$ is realized by $RB_{1:n}$.

Let $1 \leq r \leq n-1$. The rth stage of $BE_{1:(n-1)}$, called BE_r, is a pile of 2^{r-1} inverse SE stages on 2^{n-r+1} inputs/outputs each. The matrix $U_{1:(2n-1)}$ of routing tags is assigned $[C_{1:(n-1)}\ B_{1:n}]$ in Step 3 of Algorithm CNTR_BENES. Note that the matrix $[B^r_{1:n}\ C_{1:(n-1)}]$ is balanced due to the way $C_{1:(n-1)}$ is constructed in Steps 1 and 2 of Algorithm CNTR_BENES. Because the matrix $[B^r_{1:n}\ C_{1:(n-1)}]$ is balanced and $C_{1:(n-1)}$ passes $BE_{1:(n-1)}$, the following is true: the first stage of $BE_{1:(n-1)}$ partitions $B_{1:n}$ into two submatrices of 2^{n-1} rows each such that the *first* $n-1$ columns of each submatrix form a balanced matrix; similarly, each of the 2 inverse SE stages of BE_2 partitions the submatrix at its inputs into two submatrices of 2^{n-2} rows each such that the *first* $n-2$ columns of each submatrix form a balanced matrix; so, each inverse SE stage of BE_r partitions the submatrix arrived in its inputs into two submatrices of 2^{n-r} rows each such that the *first* $n-r$ columns of each submatrix form a balanced matrix. This implies that, when $[C_{1:(n-1)}\ B_{1:n}]$ is routed through $BE_{1:(n-1)}$, $[C_{1:(n-1)}\ B_{1:n}]$ is transformed into $[I_{1:(n-1)}\ B'_{1:n}]$ such that $B'_{1:n}$ fits $F^{sa}_{1:n}$. It follows from Theorem IV.1 that $B'_{1:n}$ can be realized by $RB_{1:n}$. Therefore, the Benes network realizes the permutation represented by $B_{1:n}$ when the ith input uses the ith row of $U_{1:(2n-1)}$ as the routing tag. This completes the correctness proof of Algorithm CNTR_BENES.

Steps 2 and 3 of Algorithm CNTR_BENES can be implemented using both the procedure COLUMN and procedure COLUMN_PRAM which are serial and parallel, respectively. Recall that procedure COLUMN and COLUMN_PRAM take $O(N)$ time and $O(\log N)$ time, respectively. Because the columns of $C_{1:(n-1)}$ are determined sequentially, Algorithm CNTR_BENES takes $O(N\log N)$ time in serial and $O(\log^2 N)$ in parallel. Due to the fact that it takes $O(\log N)$ time to set up the switches of the Benes network if the routing tags are available, realizing a permutation on the Benes network using the Algorithm CNTR_BENES takes $O(N\log N)$ time in serial and $O(\log^2 N)$ in parallel. As shown next, these complexities are better than those of the algorithm, proposed by Lee [13], which also sets up the switches of the Benes network stage by stage from left to right.

To describe Lee's algorithm [13], several definitions are needed. A ***Complete Residue System modulo m, CRS(mod m),*** is a set of m integers which contains

exactly one representative of each residue class mod m. A *Complete Residue Partition, CRP,* is a partition of $CRS(mod\ 2^k)$ into two CRS's$(mod\ 2^{k-1})$, $k \geq 1$. Lee's algorithm sets the switches of the rth stage, $1 \leq r \leq n-1$, of the Benes network as follows: in order to perform 2^{r-1} CRP's on the 2^{r-1} CRS's$(mod\ 2^{n-r+1})$, switches are set sequentially in such a way that every two switches that are set consecutively must have a common residue class number. According to Lee, the control algorithm for Benes network takes $O(N\log N)$ time because each stage is controlled in $O(N)$ time required for the CRP control. However, not included in the analysis are $O(N/2^{r-1})$ comparisons in determining the setting of the next switch which has a common residue class number with the switch just set up. If the comparisons are accounted for, Lee's algorithm indeed takes $O(N^2+2(N/2)^2+\cdots+(N/4)4^2) = O(N^2)$ time for serial control. For the same reason, the algorithm takes $O(N^2)$ time when the number of comparisons are also included in the complexity (instead of $O(N)$). Lee also proposed in [9] a routing algorithm for $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ (i.e., reduced $\Omega_N\Omega_N^{-1}$) which also sets up the switches using the notion of $CRS(mod\ m)$ and CRP. Therefore, this algorithm also takes $O(N^2\log N)$ time in serial and $O(N^2)$ time in parallel. The CRP done at the first stage which takes $O(N^2)$ time determines the overall complexity of the parallel algorithm.

The following example uses figures to illustrate how the Algorithm CNTR_BENES determines a routing tags matrix $U_{1:(2n-1)}$ for a given permutation on N=8 numbers.

Example IV.2. Let N=8 and $R_{8\times3}$=$[r_1\ r_2\ r_3]$. In this example, given a permutation b=(0 3 2 1 6)(4 7)(5), the matrix $U_{1:5} = [C_{1:2}\ B_{1:3}]$ of routing tags to realize the permutation is determined. The binary representation $B_{1:3} = [b_1\ b_2\ b_3]$ of b is shown in Figure IV.2. As explained in Algorithm CNTR_BENES, the matrix $C_{1:2}$ needs to be determined. Columns of $C_{1:2}$ can be determined one by one from left to right using both procedures COLUMN and COLUMN_PRAM. Column c_1 must be a vector such that matrices $[R_{2:3}\ c_1]$ and $[B_{1:2}\ c_1]$ are balanced. So, c_1 is determined by a 2-labeling of the union of the perfect matching graphs of $R_{2:3}$ and $B_{1:2}$, as shown in Figure IV.2. Similarly, c_2 is determined by a 2-labeling of the union of the perfect matching graphs of $[r_3\ c_1]$ and $[b_1\ c_1]$, so that matrices $[r_3\ c_1\ c_2]$ and $[b_1\ c_1\ c_2]$ are balanced, as shown in Figure IV.2. Using the routing scheme described in Step 3 of Algorithm CNTR_BENES, the switches of the Benes network are set to realize the permutation b, as shown in Figure IV.3. End of Example.

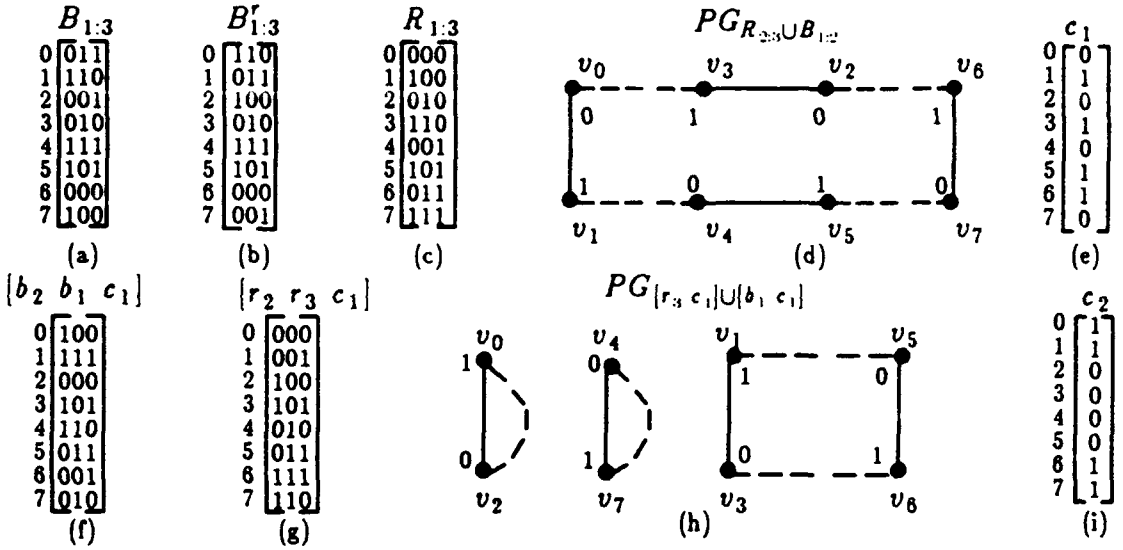


Figure IV.2. (a) $B_{1:3} = [b_1\ b_2\ b_3]$, the binary representation of a given permutation b=(0 3 2 1 6)(4 7)(5); (b) $B'_{1:3} = [b_3\ b_2\ b_1]$; (c) $R_{1:3} = [r_1\ r_2\ r_3]$; (d) $PG_{R_{2:3} \cup B_{1:2}}$, the union of the perfect matching graphs of $R_{2:3}$ and $B_{1:2}$; the solid edges (respectively, the dashed edges) form the perfect matching graph of $R_{2:3}$ (respectively, $B_{1:2}$); (e) c_1 obtained from a 2-labeling of $PG_{R_{2:3} \cup B_{1:2}}$; (f) $[b_2\ b_1\ c_1]$; (g) $[r_2\ r_3\ c_1]$; (h) $PG_{[r_3\ c_1] \cup [b_1\ c_1]}$, the union of the perfect matching graphs of $[r_3\ c_1]$ and $[b_1\ c_1]$; the solid edges (respectively, the dashed edges) form the perfect matching graph of $[r_3\ c_1]$ (respectively, $[b_1\ c_1]$); (i) c_2 obtained from a 2-labeling of $PG_{[r_3\ c_1] \cup [b_1\ c_1]}$.

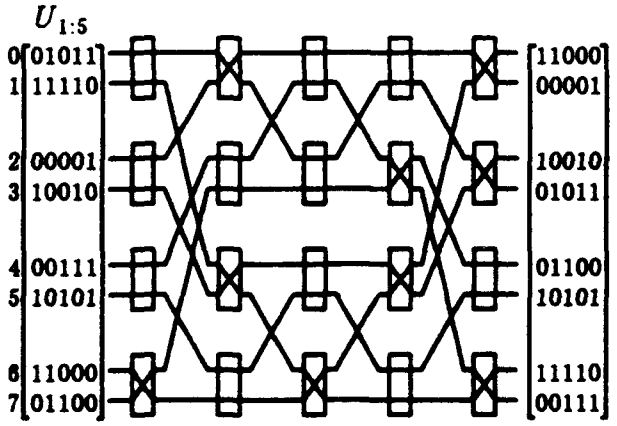


Figure IV.3. The Benes network $BS_{1:5}$ with 8 inputs/outputs; the switches are set according to the routing tags matrix $U_{1:5} = [c_1\ c_2\ b_1\ b_2\ b_3]$ using the routing scheme described in Step 3 of Algorithm CNTR_BENES.

Because any even component can be 2-labeled in 2 different ways, there exist at least $2^{(2^{p-1})}$ different choices of c_ps. This implies that the SBs of $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ has many different settings to realize a given permutation.

Theorem V.1. The network $SE_{1:(n-1)}IP^sSE_{1:n}^{-1}$ is rearrangeable.

Proof. It is first shown that all the switches at last stage of the network $SE_{1:n}$ (i.e., omega network) are set straight if the last bit of any input equals the last bit of its destination address. Let $i^m=i_1^m i_2^m \cdots i_n^m$ and $d^r=d_1^r d_2^r \cdots d_n^r$ denote the label and destination address of the mth input, respectively, where $0 \leq m,r \leq N-1$. Note that i^m is the mth row of $I_{1:n}$ because the inputs are labeled from 0 to $N-1$ in ascending order. Lawrie [1] has shown that, at any given stage k, $1 \leq k \leq n$, an input at the position $i_k^m i_{k+1}^m \cdots i_n^m d_1^r d_2^r \cdots d_{k-1}^r$ is mapped by the perfect shuffle pattern to the position $i_{k+1}^m i_{k+2}^m \cdots i_n^m d_1^r d_2^r \cdots d_{k-1}^r i_k^m$, and is then switched into the position $i_{k+1}^m i_{k+2}^m \cdots i_n^m d_1^r d_2^r \cdots d_{k-1}^r d_k^r$ by the corresponding SB. Thus, at the nth stage of $SE_{1:n}$ an input which has been switched to position $i_n^m d_1^r d_2^r \cdots d_{n-1}^r$ is mapped by the perfect shuffle pattern to $d_1^r d_2^r \cdots d_{n-1}^r i_n^m$, and is then switched into position $d_1^r d_2^r \cdots d_{n-1}^r d_n^r$. So, if $i_n^m=d_n^r$, then the switch to which i^m is an input is set straight. Therefore, if $i_n^m=d_n^r$ for every pair of input and its destination, then all the switches at the last stage are set straight, i.e., at the last stage only the shuffle pattern is needed and all the switches can be removed.

Given a balanced matrix $B_{1:n}$, there exists a balanced matrix $D_{1:n}$ determined by the repeated application of Lemma III.1 such that the matrices $[I_{1:n}\ D_{1:n}]$ and $[B_{1:n}\ D_{1:n}]$ are balanced [2]. This means that a given permutation represented by $B_{1:n}$ can be expressed as a product of permutation $D_{1:n}$ realized on $SE_{1:n}$ (Proposition 5.1 [2]) and permutation $D_{1:n} \rightarrow B_{1:n}$ realized on $SE_{1:n}^{-1}$ (Lemma V.1). It is shown below that the last column of $D_{1:n}$ can always be made equal to the last column of $I_{1:n}$, which implies that the last stage of $SE_{1:n}$ can be replaced by IP^s because it follows from the above paragraph that in this case all the switches of this stage can always be set straight. Because $[I_{1:n}\ D_{1:(n-1)}]$ is balanced, the matrix $[i_n\ D_{1:(n-1)}]$ is balanced. Because the number of columns in $[i_n\ D_{1:(n-1)}]$ is not greater than n, $[D_{1:(n-1)}\ i_n]$ is also balanced. So, the matrices $[I_{1:n}\ D_{1:(n-1)}\ i_n]$ and $[B_{1:n}\ D_{1:(n-1)}\ i_n]$ are balanced, which implies that d_n can be made equal to i_n. Because $[I_{1:n}\ D_{1:(n-1)}\ i_n]$ is balanced, the permutation represented in binary by $[D_{1:(n-1)}\ i_n]$ passes $SE_{1:n}$. Thus, the mth input of $SE_{1:n}$, denoted by i^m, is sent to the output d^r whose value equals the contents of the mth row of $[D_{1:(n-1)}\ i_n]$. When the mth row of $[D_{1:(n-1)}\ i_n]$ is used as the routing tag for the mth input of $SE_{1:n}$, all the SBs of SE_n are set straight because $d_n^r = i_n^m$. This implies that the nth bits of the routing tags are no longer needed. Therefore, when all the SBs of the last stage of $SE_{1:n}$ are removed, the permutation corresponding to $[D_{1:(n-1)}\ i_n]$ can be realized through $SE_{1:(n-1)}IP^s$ using only $D_{1:(n-1)}$ as a matrix of routing tags. Because $[B_{1:n}\ D_{1:(n-1)}\ i_n]$ is balanced, it follows from Lemma V.1 that $SE_{1:n}^{-1}$

realizes the permutation represented by $[D_{1:(n-1)}\ i_n] \rightarrow B_{1:n}$. Therefore, the permutation represented by $B_{1:n}$ is realized by the network $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$, that is, $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ is rearrangeable. □

The following algorithm determines the routing tags of the inputs of $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ to realize any permutation.

Algorithm RSE

Input: A balanced matrix $B_{1:n} = [b_1\ b_2 \cdots b_n]$ and the identity permutation matrix $I_{1:n}$.

Output: A matrix $U_{1:(2n-1)}$ such that, when its mth row, $0 \leq m \leq N-1$, is used as the routing tag for the mth input of $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$, the permutation represented by $B_{1:n}$ is realized by $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$.

Step 1. Let p denote an integer variable. Set $p=1$. Determine a column vector d_1 such that the matrices $[I_{2:n}\ d_1]$ and $[B_{2:n}\ d_1]$ are balanced.

Step 2. Increment p by 1. If $p>n-1$, go to next step; otherwise, first determine a column vector d_p such that $[I_{(p+1):n}\ D_{1:(p-1)}]$ and $[B_{(p+1):n}\ D_{1:(p-1)}]$ are balanced, and then go to Step 2.

Step 3. Let $U_{1:(2n-1)} = [D_{1:(n-1)}\ B_{1:n}]$. For $1 \leq j \leq n-1$, a SB at the jth stage of $SE_{1:(n-1)}$ examines the bit u_j of the routing tag $(u_1\ u_2 \cdots u_{2n-1})$ of its inputs, while for $n \leq j \leq 2n-1$ a SB at the jth stage of $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ examines the bit u_{3n-j+1} of the routing tag $(u_1\ u_2 \cdots u_{2n-1})$ of its input. **Stop.**

To prove the correctness of Algorithm RSE, note that it follows from Algorithms CONS_COLUMN and CONS_COLUMN_PRAM that Step 1 and Step 2 of Algorithm RSE are realizable. Theorem V.1 has shown that $B_{1:n}$ passes $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$. Now, it remains to show that the mth row of $U_{1:(2n-1)} = [D_{1:(n-1)}\ B^r_{1:n}]$ can be used as the routing tag for the mth input of $SE_{1:(n-1)}IP^sSE^{-1}_{1:n}$ to realize $B_{1:n}$.

Let $U_{1:(2n-1)}(m)$ denote the mth row of $U_{1:(2n-1)}$. The destination tag routing scheme used in $SE_{1:k}$, $k \geq 1$, (respectively, $SE^{-1}_{1:k}$) is as follows: at the jth stage, $1 \leq j \leq k$, of $SE_{1:k}$ (respectively, $SE^{-1}_{1:k}$) a SB examines the u_j (respectively, the u_{k-j+1}) of the routing tag $(u_1 u_2 \cdots u_k)$ of its input [1]. If $u_j=0$, the input is sent to the upper output of the SB; otherwise, the lower output of the SB is taken. Therefore, the routing scheme described in Step 3 is used, and $U_{1:(2n-1)}$ is assigned $[D_{1:(n-1)}\ B_{1:n}]$. This completes the correctness proof of Algorithm RSE.

Algorithms CNTR_BENES and RSE are the same, except that Algorithm RSE replaces $R_{1:n}$ by $I_{1:n}$. Therefore, the complexities of these algorithms are

the same, that is, Algorithm RSE also takes $O(N\log N)$ time in serial and $O(\log^2 N)$ time in parallel.

The following example uses figures to illustrate Algorithm RSE.

Example V.1. Let N=8 and $I_{8\times 3}$=$[i_1\ i_2\ i_3]$. In this example, given a permutation b=(0 3 2 1 6)(4 7)(5), the matrix $U_{1:5} = [D_{1:2}\ B_{1:3}]$ of routing tags to realize the permutation on $SE_{1:2}IP^sSE_{1:3}^{-1}$ is determined. Columns of $D_{1:2}$ can be determined one by one from left to right using both procedures COLUMN and COLUMN_PRAM. Column d_1 must be such that $[I_{2:3}\ d_1]$ and $[B_{2:3}\ d_1]$ are balanced. So, d_1 is determined by a 2-labeling of the union of the perfect matching graphs of $I_{2:3}$ and $B_{2:3}$, as shown in Figure V.1. Similarly, d_2 is determined by a 2-labeling of the union of the perfect matching graphs of $[i_3\ d_1]$ and $[b_3\ d_1]$, so that matrices $[i_3\ d_1\ d_2]$ and $[b_3\ d_1\ d_2]$ are balanced, as shown in Figure V.1. Using the routing scheme described in Step 3 of Algorithm RSE, the switches of the $SE_{1:2}IP^sSE_{1:3}^{-1}$ network are set to realize the permutation b, as shown in Figure V.2. End of Example.

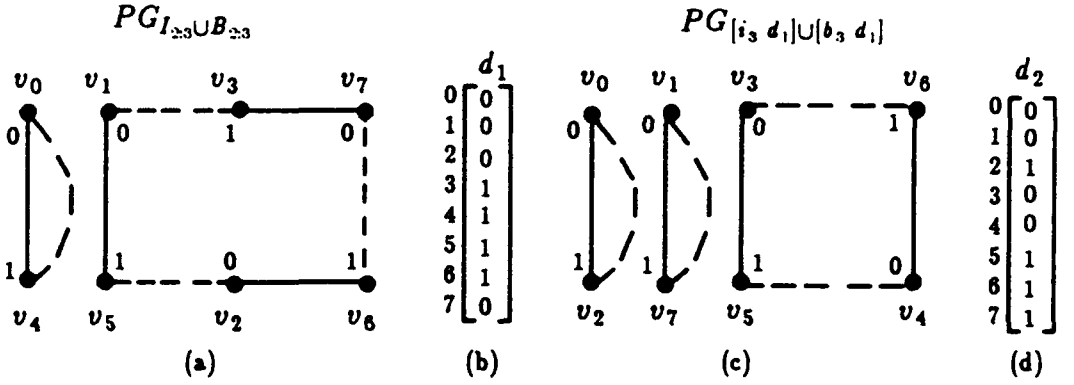


Figure V.1. **(a) $PG_{I_{2:3}\cup B_{2:3}}$, the union of the perfect matching graphs of $I_{2:3}$ and $B_{2:3}$; the solid edges (respectively, the dashed edges) form the perfect matching graph of $I_{2:3}$ (respectively, $B_{2:3}$); (b) d_1 obtained from a 2-labeling of $PG_{I_{2:3}\cup B_{2:3}}$; (c) $PG_{[i_3\ d_1]\cup[b_3\ d_1]}$, the union of the perfect matching graphs of $[i_3\ d_1]$ and $[b_3\ d_1]$; the solid edges (respectively, the dashed edges) form the perfect matching graph of $[i_3\ d_1]$ (respectively, $[b_3\ d_1]$); (d) d_2 obtained from a 2-labeling of $PG_{[i_3\ d_1]\cup[b_3\ d_1]}$.**

Because Algorithm RSE is the same as Algorithm CNTR_BENES except that $R_{1:n}$ is replaced by $I_{1:n}$, the minimum number of settings that enables a $B_{N\times n}$ to pass through $SE_{1:(n-1)}IP^sSE_{1:n}^{-1}$ is also equal to $\prod_{p=1}^{n-1} 2^{(2^{p-1})}$.

VI. CONCLUSIONS

New routing algorithms are presented in this paper for realizing any permutation in rearrangeable networks Benes and the reduced $\Omega_N\Omega_N^{-1}$. The proposed routing algorithms in this paper are easier to comprehend and they take $O(N\log N)$ time in serial and $O(\log^2 N)$ time in parallel. These algorithms first compute a routing tag of $2n-1$ bits for every input, then set the switches one stage at a time such that the switches at the jth stage are set by decoding the jth bits of the pre-computed routing tags at their inputs. The last n bits of any routing tag are the destination address bits. The routing algorithm of the reduced $\Omega_N\Omega_N^{-1}$ follows from a new rearrangeability proof presented in this paper.

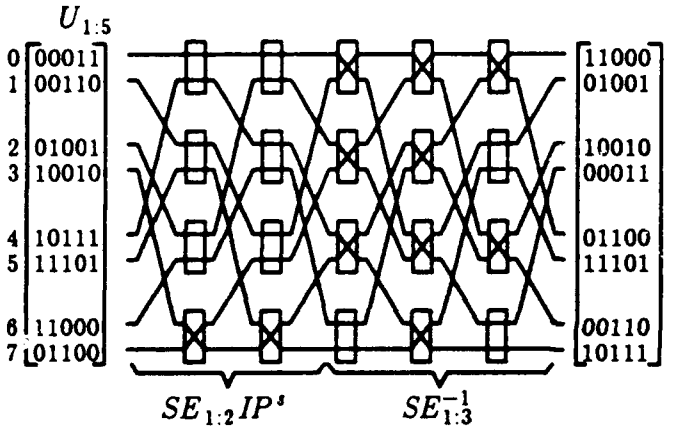


Figure V.2. The $SE_{1:2}IP^sSE_{1:3}^{-1}$ network with 8 inputs/outputs; the switches are set according to the routing tags matrix $U_{1:5} = [d_1\ d_2\ b_1\ b_2\ b_3]$ using the routing scheme described in Step 3 of Algorithm RSE.

VII. APPENDIX

Procedure CONS_LIST(N) constructs a linear linked list of N nodes, called H, each of which contains 2 fields, namely, LEFT and RIGHT which are the pointers to the previous and the following node, respectively. For $1 \leq i \leq N-1$, the LEFT field of the ith node of H equals $i-1$. For $0 \leq i \leq N-2$, the RIGHT field of the ith node of H equals $i+1$. Let p be the pointer to the header of H and initialized to 0. Procedure DELETE(k) deletes the node $H(k)$ from H. In addition, it updates the pointer if $H(k)$ is the first node of the current H.

```
procedure CONS_LIST(N)
 Let p be the header of H and p := 0
 LEFT(0) := nil; RIGHT(0) := nil
 for s := 1 to N−1 do LEFT(s) := s−1; RIGHT(s) := nil; RIGHT(s−1) := s
 end
end CONS_LIST(N)

procedure DELETE(k)
 Let p be the header of H and p := 0
 if k=p then do p := RIGHT(p); LEFT(p) := nil; end
            else RIGHT(LEFT(k)) := RIGHT(k)
end DELETE(k)
```

Proof of Theorem IV.2. (←) It is shown that if $D_{N\times k}$ fits $F^{sq}_{N\times k}$ then $[R_{N\times n}\ D_{N\times k}]$ is balanced. By definition, a matrix with N rows and more than n columns is balanced if every n consecutive columns form a balanced matrix. Therefore, it suffices to show that for any j, $1 \le j \le k$, $[R_{(j+1):n}\ D_{1:j}]$ is balanced. Partition $[R_{(j+1):n}\ D_{1:j}]$ into 2^{n-j} submatrices $S^h_{2^j\times n} = [R^h_{(j+1):n}\ D^h_{1:j}]$, $0 \le h \le 2^{n-j}-1$, such that any of $S^h_{2^j\times n}$, $R^h_{(j+1):n}$, and $D^h_{1:j}$ contains 2^j rows whose indices belong to the same block of the partition P^*_j, that is, the row indices of $S^h_{2^j\times n}$, $R^h_{(j+1):n}$, and $D^h_{1:j}$ consist of the numbers $(h\times 2^j)$ to $[(h+1)\times 2^j]-1$ inclusive. $D^h_{1:j}$ has 2^j distinct rows because it is a balanced matrix of order $(2^j\times j)$. Therefore, no matter what the rows of $R^h_{(j+1):n}$ are, the 2^j rows of S^h are distinct. On the other hand, because of the definitions of $R_{N\times n}$ and P^*_j, the rows of $R^h_{(j+1):n}$ are all identical and are distinct from any other row of $R_{(j+1):n}$. It follows that there exist no common rows in any two different submatrices $S^x_{2^j\times n}$ and $S^y_{2^j\times n}$, where $x\neq y$ and $0 \le x, y \le 2^{n-j}-1$. Therefore, there are 2^{n-j} submatrices $S^h_{2^j\times n}$ that have no common rows and each submatrix has 2^j distinct rows. So, the total number of distinct rows in $[R_{(j+1):n}\ D_{1:j}]$ equals $2^n = 2^{n-j}\times 2^j$. Because $[R_{(j+1):n}\ D_{1:j}]$ is of order $N\times n$, by definition II.2.1 the matrix $[R_{(j+1):n}\ D_{1:j}]$ is balanced.

(→) It is shown that if $[R_{N\times n}\ D_{N\times k}]$ is balanced then $D_{N\times k}$ fits $F^{sq}_{N\times k}$. Consider the balanced matrix $[R_{(j+1):n}\ D_{1:j}]$. By definition of $R_{N\times n}$, the rows of $R^h_{(j+1):n}$ of order $(2^j\times j)$ (whose indices belong to the same block in P^*_j) are identical. By hypothesis, $[R_{N\times n}\ D_{N\times k}]$ is balanced and, therefore, $[R_{(j+1):n}\ D_{1:j}]$ must also be balanced. This implies that $D^h_{1:j}$ (whose indices belong to the same block of P^*_j) must be a balanced matrix of order $2^j\times j$ (because, as mentioned above, the rows in $R^h_{(j+1):n}$ are identical). This is true for all $j=1, \ldots, k$ and all blocks of P^*_j. Therefore, by the definition of fit (Definition II.2.9) and the definition of standard frame $F^{sq}_{N\times k}$ (Definition II.2.7), $D_{N\times k}$ fits $F^{sq}_{N\times k}$. □

Theorem IV.2 establishes a relation between the reverse permutation matrix $R_{N\times n}$ and $F^{sa}_{N\times k}$, $1 \leq k \leq n$. The basic idea of Theorem IV.2 is illustrated next in an example.

Example IV.1. Let N=16 and n=4. The reverse permutation matrix $R_{16\times 4}$, a balanced matrix $D_{16\times 4}$ that fits $F^{sa}_{16\times 4}$, and the matrix $[R_{16\times 4}\ D_{16\times 4}]$ with the frame $F^{sa}_{16\times 4}$ are illustrated in Figure IV.1. As it is seen from $[R_{16\times 4}\ D_{16\times 4}]$, the rows of $R^h_{(j+1):n}$ are all identical and are distinct from any other row of $R_{(j+1):n}$. Also, notice that $D^h_{1:j}$ is a balanced matrix of order $2^j\times j$, so that each row in it is distinct. End of example.

	(a)		(b)		(c)				
						f^{sa}_1	f^{sa}_2	f^{sa}_3	f^{sa}_4
0	0000	0	0000	0	0000	0	0	0	0
1	1000	1	1100	1	1000	1	1	0	0
2	0100	2	0100	2	0100	0	1	0	0
3	1100	3	1011	3	1100	1	0	1	1
4	0010	4	1000	4	0010	1	0	0	0
5	1010	5	0010	5	1010	0	0	1	0
6	0110	6	0111	6	0110	0	1	1	1
7	1110	7	1110	7	1110	1	1	1	0
8	0001	8	0011	8	0001	0	0	1	1
9	1001	9	1001	9	1001	1	0	0	1
10	0101	10	1111	10	0101	1	1	1	1
11	1101	11	0110	11	1101	0	1	1	0
12	0011	12	0101	12	0011	0	1	0	1
13	1011	13	1010	13	1011	1	0	1	0
14	0111	14	0001	14	0111	0	0	0	1
15	1111	15	1101	15	1111	1	1	0	1

Figure IV.1. (a) The reverse permutation matrix $R_{16\times 4}$ for N=16. (b) A $D_{16\times 4}$ that fits $F^{sa}_{16\times 4}$. (c) $[R_{16\times 4}\ D_{16\times 4}]$ with $F^{sa}_{16\times 4}$.

REFERENCES

[1] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Comput.*, vol. c-24, pp. 1145-1155, Dec. 1975.

[2] N. Linial and M. Tarsi, "Interpolation between bases and the Shuffle-Exchange network," *Europ. J. Combinatorics*, vol. 10, pp. 29-39, 1989.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, New York, 1979.

[4] L. Lovasz and M. D. Plummer, *Matching Theory*, Annals of Discrete Math., 29, North-Holland, 1986.

[5] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. on Comput.*, vol. c-20, pp. 153-161, Feb. 1971.

[6] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[7] A. Waksman, "A permutation network," *Journal of the ACM*, vol. 15, No. 1, pp. 159-163, Jan. 1968.

[8] C. P. Kruskal and M. Snir, "A unified theory of interconnection network structure," *Theoretical Computer Science*, vol. 48, pp. 75-94, 1986.

[9] K. Y. Lee, "On the rearrangeability of 2(log N)-1 stage permutation networks," *IEEE Trans. Comput.*, vol. c-34, pp. 412-425, May 1985.

[10] C. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. on Comput.*, vol. c-29, pp. 696-702, 1980.

[11] T. Lang, "Interconnections between processors and memory modules using the Shuffle Exchange network," *IEEE Trans. on Comput.*, vol. c-25, No 5, May 1976.

[12] T. Etzion and A. Lempel, "An efficient algorithm for generating linear transformations in a shuffle-exchange network," *SIAM J. Comput.*, Vol. 15, No. 1, pp. 216-221, Feb. 1986.

[13] K.Y. Lee, "A new Benes network control algorithm," *IEEE Trans. on Comput.*, vol. c-36, pp. 768-772, June 1987.

[14] D. Nassimi and S. Sahni, "A self-routing Benes network and parallel permutation algorithms," *IEEE Trans. on Comput.*, vol. c-30, pp. 332-340, May 1981.

[15] ———, "Parallel algorithms to set up the Benes permutation network," *IEEE Trans. on Comput.*, vol. c-31, pp. 148-154, Feb. 1982.

[16] D.C. Opferman and N.T. Tsao-Wu, "On a class of rearrangeable switching networks, Part I: Control algorithm," *Bell System Tech. J.*, vol. 50, pp. 1579-1600, 1971.

[17] H. Cam, *Design and permutation routing algorithms of rearrangeable networks*, Ph.D. Thesis, Purdue Univ., May 1992.

[18] H. Cam and J.A.B. Fortes, "Frames: a simple characterization of permutations realized by frequently used networks," submitted for publication.

[19] J.J. Rotman, *The theory of groups: An introduction*, Allyn & Bacon, Rockleigh, N.J., 1965.

[20] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, Lexington Books, 1986.

[21] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.